Generalized-Mapping IPC for L4

Andreas Weigand

Technischen Universität Dresden
Department of Computer Science
Operating Systems Group

April 2004

Contents

1 Introduction **9**
1.1 Organization of this Document . . . 9
1.2 Acknowledgements . . . 9

2 Background and Related Work **11**
2.1 Microkernels . . . 11
2.2 Access Control . . . 12
2.2.1 Access Matrix . . . 12
2.2.2 Access Control Lists . . . 12
2.2.3 Capabilities . . . 13
2.3 The L4 Memory Model . . . 14
2.3.1 Recursive Virtual Address Space Construction . . . 15
2.3.2 Mapping Database . . . 16
2.3.3 Page Fault Handling . . . 17

3 Design **19**
3.1 Requirements . . . 20
3.2 Generalized Mappings for IPC Management . . . 20
3.2.1 Distribution of Send Capabilities . . . 21
3.2.2 Preventing the Unwanted Distribution of Rights . . . 22
3.2.3 Capability Fault Handling . . . 24
3.2.4 Local Id's on the Sender Side . . . 25
3.2.5 Local Id's on the Receiver Side . . . 25
3.2.6 Receiver Selection . . . 27
3.2.7 Starting a System . . . 28
3.3 Scenarios . . . 29
3.3.1 A Message Interceptor . . . 29
3.3.2 Global Id's . . . 31
3.4 The Prototype . . . 33
3.4.1 General Architecture . . . 33

4 Implementation **35**
4.1 Library Architecture . . . 35
4.2 IPC Implementation in Detail . . . 36
4.2.1 Receiver Preparation . . . 36
4.2.2 Send Operation in Detail . . . 37
4.2.3 Implementing an Atomic Call . . . 38
4.2.4 Implementing Timeouts for IPC . . . 39
4.2.5 Capability Mapping and Unmapping . . . 40
4.2.6 Capability Fault Handling . . . 40
4.3 Matching Requirements . . . 41
4.4 Receiver Selection Algorithms . . . 41

4.4.1 Linear List Search 41
4.4.2 Binary Trees and Tries 42
4.4.3 Page Table like Structures 44
4.4.4 Hashing 46

5 Evaluation **49**
5.1 Performance on the Sender Side 50
5.2 Performance on the Receiver Side 51

6 Conclusion **53**
6.1 Future Work 54

A Appendix **55**
A.1 Glossary 55
A.2 Acronyms 56

List of Figures

2.1 A cryptographically-protected Amoeba capability. 13
2.2 Memory management policies in a monolithic kernel. 14
2.3 Memory management policies in a microkernel-based system. 14
2.4 A page is mapped using IPC. 15
2.5 An unmap example. 16
2.6 Page fault processing in L4. 17

3.1 A subsystem with a defined information flow. 23
3.2 The same subsystem with unwanted communication channels. 23
3.3 The unwanted distribution of rights. 24
3.4 Acting on Behalf. 27
3.5 Three tasks, client can communicate directly to the server. 30
3.6 After inserting the message interceptor. 30
3.7 An encoding for global Id's. 31
3.8 An example for tasks using global Id's for communication. 32

4.1 A system with GM tasks and non-GM tasks. 35
4.2 The distributor thread. 36
4.3 The receiver prepares for message reception. 37
4.4 Control flow for a GM-IPC send operation. 37
4.5 Control flow for a GM-IPC call operation. 38
4.6 Page table structure of the Pentium MMU. 44
4.7 A possible page table structure for receiver selection. 46

List of Tables

5.1 Time needed on sender side in cycles. 51
5.2 The time for receiver selection in processor cycles. 52

Chapter 1

Introduction

For years operating systems were built using a monolithic kernel containing all needed functionality in one large piece of software. In contrast, the microkernel development, started in the 1980s, focuses on minimizing the kernel to a minimal set of abstractions and implement all remaining operating-system functionality as servers at user level. Ideally, a microkernel provides only address spaces (protection domains), threads (the active entities inside an address space), and communication between threads (*inter-process communication*, IPC).

Servers and applications implemented on top of the microkernel execute in their own address space. Communication between servers and applications uses the kernel-provided IPC mechanism. Therefore the performance and functionality of the IPC mechanism is important for microkernel-based systems.

System-wide security policies require the control of communication relationships. Certain constraints on the flow of information are imposed and need to be enforced. This means that both the direct communication of task and their access rights to objects need to be controlled. The absence of the latter would allow for indirect communication via a shared resource. To achieve the aforementioned objectives a semantically well defined, general and efficient control mechanism is needed. In this thesis I refined and implemented such a mechanism.

Generalized Mappings is devised upon the ideas of the L4 memory management scheme. Resources are referred to by local names only and access rights can be transferred. The coverage of new notions as send rights required the extension of the existing name space. Originally only virtual addresses were used as name space. The existence of objects beyond page frames required an additional naming scheme. Capabilities as opaque handles to objects of various classes fulfill this role. The manipulating operations (map, grant, unmap) can be applied the same way as for memory objects.

This thesis refined an initially vague model and evaluated its appropriateness on a prototypical implementation. The results confirmed its feasibility.

1.1 Organization of this Document

This thesis is organized as follows. In the next chapter I describe the background of this thesis. I introduce microkernels, access-control mechanisms and the L4 memory model. In Chapter 3, I present the design of the Generalized-Mapping IPC mechanism. First, I describe the requirements for the IPC mechanism, then the mechanism itself. Chapter 4 explains the prototypical implementation. Furthermore, I discuss algorithms for receiver selection. I evaluate the performance of the new IPC mechanism in Chapter 5. Finally, I conclude this thesis in Chapter 6 and give an outlook on future work.

1.2 Acknowledgements

This thesis would have been impossible without the support of many people. I am deeply indebted to all of them. First of all I would like to thank Prof. Hermann Härtig. He was the driving force behind the ideas of Generalized Mappings. My special thanks go to my supervisors Christian Helmuth and Marcus Völp for their guidance and the support they gave me during the work on this thesis. I had numerous interesting discussions with them. Finally thank Michael Peter, Udo Steinberg and Marcus Völp for proofreading this thesis. Their valuable comments and feedback helped me to improve this document.

Chapter 2

Background and Related Work

2.1 Microkernels

In operating systems, the term *kernel* denotes the fundamental part used by all other software. The kernel is special from other software, as it runs in a special CPU mode: the kernel mode. Thus it can use all processor features such as programming the memory management unit or switching tasks. Software running in user mode cannot perform such security-critical operations.

Many operating systems, such as Linux [22] or Windows [31], are based on a monolithic kernel. Such a kernel contains all operating-system services, including file systems, device drivers, network stacks, memory management, scheduling and others, packed together into a single kernel. Thereby all these services run in kernel mode. One malfunctioning or hostile component can corrupt and even crash the whole system.

In contrast, the key idea of microkernels is to provide only a minimal set of abstractions in the kernel and implement all additional functionality as servers at user level. Ideally, the microkernel implements only address spaces, inter-process communication (IPC) and basic scheduling [20]. All remaining operating system components and services, even device drivers, are user-level programs. Because each of these components executes in its own address space, they are protected from each other. This design allows for more robust operating systems because a malfunctioning service does not hamper the remaining components. Furthermore, this design approach helps to reduce the trusted computing base (TCB) to the hardware, the microkernel and basic services, such as a basic memory manager, a screen driver, a keyboard driver, and perhaps a disk driver and a file system.

The first generation of microkernels and its flagship Mach [1], however, suffered from flexibility and performance problems. Mach was a refactored Unix kernel and it was not really small (more than 140 system calls, more than 300 KB code). It provided a rich featured IPC mechanism with authorization, message buffering and a complex message format. Mach also invented the innovative concept of external pagers that allows to implement memory management strategies outside the kernel. However, due to the enormous size and the sophisticated IPC mechanism, the Mach microkernel had poor performance.

The designers of the second generation of microkernels learned from the errors of the first generation and designed their kernel for achieving performance. Liedtke constructed his L4 microkernel from scratch with the goals *IPC performance is the Master* [15] and minimality [19]. This design methodology leads to small-sized kernels with high performance.

The L4 Microkernel

The original L4 microkernel interface [18] was designed for the x86 architecture and written completely in assembly language [17]. L4 provides only three abstractions: address spaces (protection domains), threads (the executing entities in an address space) and synchronous inter-process communication (IPC) between threads. Furthermore, the kernel provides a scheduler with multiple fixed-priority levels, whereby threads with the same priority are scheduled in a round-robin fashion.

All software developed during this thesis runs on top of the Fiasco microkernel, an implementation of the L4 interface developed by Michael Hohmuth at TU-Dresden [7, 4]. Fiasco is fully binary compatible to the original L4 kernel. Furthermore, Fiasco provides a powerful in-kernel debugger and a port to the linux system-call interface [5]. Therefore, Fiasco is an excellent basis for software development for L4.

2.2 Access Control

Access Control is a key mechanism for constructing trusted secure operating systems. The access-control mechanism enforces the defined access-control policy of a system. The access-control policy defines and restricts the ability of a *subject* to invoke certain operations on an *object*.

Objects are the resources of the system. These objects can be physical resources, such as memory, CPUs, devices and other hardware, or logical resources, such as processes, communication channels, files or system calls. Certain operations can be invoked on objects depending on the object type. For example, files can be read and written to, but system calls can only be executed. Therefore, objects need to be protected [13] from unauthorized or invalid accesses.

Subjects are the active entities that access objects. In computer systems, subjects are usually processes acting on behalf of a user. Subjects can have access rights on objects to access and use these objects. However, subjects can also be potentially untrusted, for example user-supplied or downloaded program code.

So a way is needed to prohibit subjects from accessing objects that they are not authorized to access. The provided mechanism must also allow for restricting the allowed operations a subject can use on an object. For example, a task T may be authorized to read a file F, but not to write to that file. The access-control mechanism must ensure that the desired access-control policy cannot be bypassed. A subject's access to an object must not succeed if that access violates the policy.

The relationship between subjects and objects regarding the object access can be represented with the triple (subject, object, rights). Such a triple specifies the subject's rights on an object. These triples are defined by the system's access-control policy and are used by the access- control mechanism to enforce the policy.

2.2.1 Access Matrix

A possible way to store the access triples is the access matrix. The access-matrix rows consist of the subjects; the columns are the objects. An entry inside the matrix controls the access rights from a subject to an object. Whenever a subject tries to access an object, the access control mechanism checks the matrix entry whether the requested operation is allowed or not.

The main problem with the access matrix is its enormous size: The matrix needs one entry per (subject, object) pair. Also the matrix is usually sparsely populated (most subjects need and have only access to some few objects). So storing the entire, mostly empty, matrix would waste memory and disk space. Therefore, the idea is to store only the relevant parts of the matrix. Two methods are common: Storing the nonempty elements by column or storing them by row. The first method is called an *access control list*. The second method is called a *capability*.

2.2.2 Access Control Lists

An access control list (ACL) consists of all nonempty elements from a matrix column. Every ACL entry is a tuple (subject, rights). The ACL is stored together with the object it belongs to and specifies which subject can access that object in which way.

When a subject tries to invoke an operation on an object, the object's ACL is traversed to check whether the subject may invoke the requested operation or not. If more than one entry matches the subject, the first entry found is used. Therefore the access control list must be kept sorted by some metric. A common metric, used in the Unix file system, is to store entries regarding individual users at the beginning, followed by entries regarding groups of users (sometimes called roles). Default entries are stored at the end. With this metric a particular user can have less rights than a group it is in.

Since ACLs are stored together with the object, it is easy to figure out which subject can invoke which operations on that object. Only the ACL has to be analyzed for that. The revocation of rights is also

Server	Object	Rights	f(Object,Rights,Check)

Figure 2.1: A cryptographically-protected Amoeba capability.

uncomplicated: Now unwanted operations are removed from every subject's ACL entry. To revoke the access of a subject completely, the ACL entry is removed. However, care has to be taken when a subject can act also with a role account (e. g., a Unix group). Then the subject's entry must not be removed. Instead it must allow for no operation (remember also the metric described afore).

A problem with ACLs is their limited ability for access delegation. For each subject wanting access (temporarily) a new entry has to be added. After accessing the entry has to be removed again.

Access control lists are commonly used in file systems, such as the Unix file system, or the Windows 2000 file system.

2.2.3 Capabilities

When storing the rows from access matrix we talk about *capability lists* consisting of *capabilities*. A capability is a tuple (object, right) assigned to a subject.

The capability grants its owner certain access rights on an object. Thereby the object used is encoded in the capability; the subject specifies only the capability to use and the operation. To check whether the requested operation is allowed or not only the capability is inspected. No list traversing as with ACLs is needed.

Because capabilities are given to subjects, which are potentially malicious, they must be protected from tampering. Three different methods are known: First, capabilities can be protected by hardware. The Cambridge CAP Computer [24] uses different segment types for data segments and capability segments. The segments can be accessed only with instructions belonging to the segment type. Arithmetical and logical instructions can be used only on data segments. Capability related instructions work only on capability segments.

Second, capabilities can be protected by the operating system. Here only the operating system itself can access the capabilities directly, user applications only receive a handle, a local name, for the capability. A common example are file descriptors in Unix: The process uses a handle (the file descriptor) to invoke operations (read, write, seek) on a file. In Hydra [32] and in EROS [25] capabilities are protected by the operating-system kernel.

The third way is to store capabilities in user space but protect them from tampering using cryptography. This approach is used in distributed systems such as Amoeba [29]. An Amoeba capability [28], shown in Figure 2.1, encodes the server the capability belongs to, an object there and the access rights to that object. The last field is the result of a cryptographically strong one-way hash function whereby *Check* is a secret only known to the server. When the capability owner wants to access the object, it sends the request and the capability to the server. The server recalculates the hash using the object and rights fields from the transmitted capability and its internally stored check. Object access is only granted when the recalculated hash value matches the hash value from the capability. Not matching hashes indicate an altered or even a self-created capability. In this case the access is denied and the request is discarded.

Using capabilities allow for an elegant way of access delegation: One subject S can give a capability it owns to another subject R. Thereby S can restrict R's access rights to the object the capability references to a subset of its own access rights. Afterwards R can also access the object using its capability copy.

However, because capabilities are given to a subject and that subject can give copies to other subjects, the revocation of granted access rights (capabilities) is difficult. It is hard to find all outstanding copies, especially in distributed systems as Amoeba. One approach is to have an indirection object: Rather than pointing to the object itself, all capabilities point to the indirection object. Capability revocation is done by destroying the indirection object. This object becomes invalid, and thus all capabilities regarding to it too. EROS provides this method; the indirection objects are called *wrappers* there.

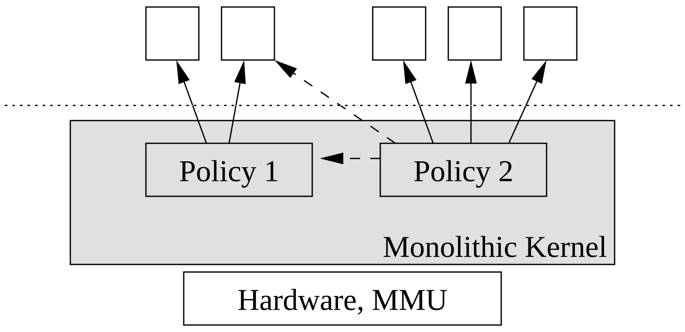

Figure 2.2: Memory management policies in a monolithic kernel.

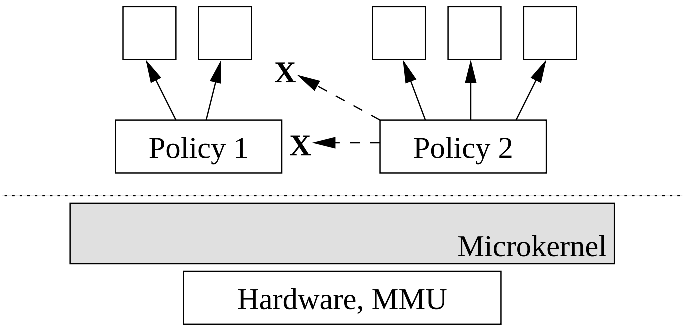

Figure 2.3: Memory management policies in a microkernel-based system.

Another way for revocation is possible in the Amoeba scheme: A server can change the check field it stores with the object. Immediately, all existing capabilities encoded with the old check value become invalid. However, neither of these mechanisms allow to revoke capabilities selectively from one subject but not from another.

2.3 The L4 Memory Model

In this section I will introduce the L4 memory model, the ideas behind it and the provided mechanisms.

The separation of address spaces to protect tasks from each other is important in operating systems as it is necessary for fault isolation and data protection. The basic mechanisms for address-space separation are provided by the hardware's memory management unit (MMU). Since programming the MMU is critical for system security, it can only be done in kernel mode but not in user mode. Therefore the kernel has to program the MMU and the structures used by the MMU.

In monolithic kernels the kernel does the entire memory and address space management. The kernel is part of the TCB, and because the kernel must be trusted anyway, this implementation is safe from a security perspective. The kernel also defines the policy used for memory management. However, kernel-defined policies are rather static, the applications can at most select from the provided strategies. Implementing additional application-specific management policies requires a kernel change to insert a new policy into the kernel. Another problem with this approach is the lack of isolation between the different provided policies: One malfunctioning policy module can harm the other modules and their client applications. This problem is illustrated with dashed lines in Figure 2.2.

The microkernel approach addresses these problems. Here only the fundamental management, the programming of the hardware, is done by the kernel. The kernel exports an interface for modifying the memory management structures to user level. The management policy is implemented purely at user level using this kernel-provided interface. So any required memory management strategy can be implemented

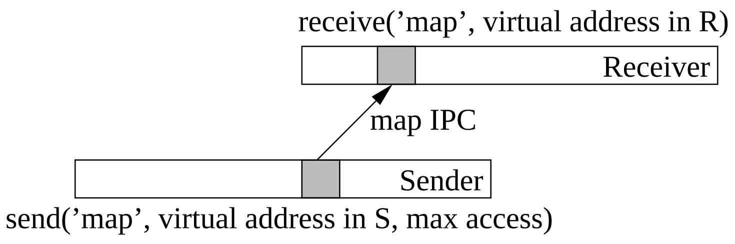
receive('map', virtual address in R)
Receiver
map IPC
Sender
send('map', virtual address in S, max access)

Figure 2.4: A page is mapped using IPC.

at user level. New policy implementations do not require a kernel change. Because of the address-space separation, a faulty memory-management policy provider can only harm its clients but not other policy providers and their clients, see Figure 2.3.

With this approach it is even possible to stack multiple policy providers: For example, a base memory provider hands out pages using a cache-coloring scheme. The next-level provider then does demand paging for its clients. In the L4 microkernel the scheme of recursive address-space construction is used for stacked memory managers.

2.3.1 Recursive Virtual Address Space Construction

The L4 kernel provides mechanisms for the recursive construction of address spaces outside the kernel [17]. These mechanisms allow a thread from one address space to give access to its pages to another thread in another address space.

The kernel enforces the security restrictions on this operation: Only pages the sender has access to can be given to the receiver. This restriction prevents from gaining access to arbitrary memory pages. It is achieved by using virtual page numbers for the operation. The other important restriction is that the source thread can grant at most the rights it has itself on a page to the receiver. So privilege elevation, for example upgrading a page from read-only to read-write, is prevented. However, downgrading the access rights, for example from read-write to read-only, is an allowed operation. Hence, the L4 model for constructing address spaces is safe because the kernel enforces the aforementioned restrictions during page transfer and because only the kernel programs the MMU hardware.

The recursive address space construction starts from an initial address space called σ_0. The kernel creates this address space with an idempotent mapping from virtual to physical pages during system start. It owns all memory except the parts the kernel uses itself. All other address spaces are created empty.

To recursively construct address spaces the kernel provides the three operations *map*, *grant* and *unmap*.

Map and Grant

A thread from an address space can *map* any of its pages to another address space if the recipient agrees. The mapped pages are inserted into the recipients address space. Afterwards the mapped pages are accessible in both the mapper's address space and the recipient's address space.

A thread from an address space can also *grant* any of its pages to another address space if the recipient agrees. The granted pages are inserted into the recipients address space. However, in contrast to map, the granted pages are removed from the granter's address space.

Both operations, map and grant, require an agreement between the source thread and the recipient thread. Therefore these operations are implemented using IPC: The sender specifies the map or grant option inside the message and provided the virtual address of the page to send and the receiver's maximum access rights to that page. Remember, the kernel maps or grants at most the actual rights from the sender. The receiver waits for a map message and specifies the virtual address a received page is mapped or granted to in its address space. This map IPC is illustrated in Figure 2.4.

With the map and the grant operation, L4 has an elegant way to delegate access rights to memory pages from one address space to another.

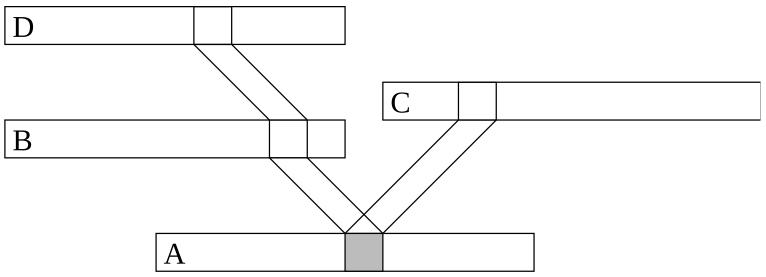

Figure 2.5: An unmap example.

Unmap

A thread from an address space may *unmap* any of its pages at any time. Afterwards, the unmapped page remains accessible in the thread's address space but is removed from all address spaces the thread (or another thread from the same address space) has mapped the page to. Thereby, unmap works recursively and removes the page also from all address spaces that received the page indirectly. This is illustrated in Figure 2.5: The page is revoked from all address spaces that received it directly (B, C) or indirectly (D) from A.

In contrast to the map and grant operations unmap does not require an explicit agreement with the address spaces the pages are removed from. Such an agreement would allow a DoS attack against threads from the unmapping address space when threads that received the page are not willing to agree on the unmap. Nevertheless, the operation is safe because it is restricted to owned pages as map and grant are too. The recipients of mapped pages already agreed on a potential unmap when accepting the received the page.

The unmap operation allows also to revoke access rights from mapped pages: The unmapping thread specifies the maximum allowed access rights for the page. For example, a read-write page can be downgraded to read-only. Revoking all access rights results in removing the page completely as described at the beginning. The unmap mechanism allows for revocation of previously delegated (mapped) pages.

2.3.2 Mapping Database

In L4 address spaces are recursively constructed and managed at user level using the kernel provided map, grant and unmap operations. These operations allow for arbitrary user-supplied paging and memory management strategies that are in no way restricted by the microkernel.

The map and grant operations work with virtual addresses and can be implemented by copying the page-table entries and applying further restrictions (e. g., setting the read-only flag). For unmap, however, additional information is required: The kernel has to unmap the pages from all address spaces that received the pages directly or indirectly from the unmapping thread. To do that the kernel needs the address spaces and the virtual addresses within where the page has been mapped to. This information is stored in the *mapping database*.

For each virtual page existing in an address space the mapping database contains an entry. During a map operation the receiver address space and the virtual address within are recorded in the mapping database entry for the mapped page. So the unmap operation can iterate through the database entries to find all directly or indirectly derived mappings of a page.

Pages are mapped from one task to another task. At any time a page can origin from at most one task. Further on, the kernel prevents cyclic mapping of a page. Therefore the mapping database for any page is a directed acyclic graph. Because all page mappings start at the initial address space σ_0, the mapping database for a page can be interpreted as a tree rooted in σ_0.

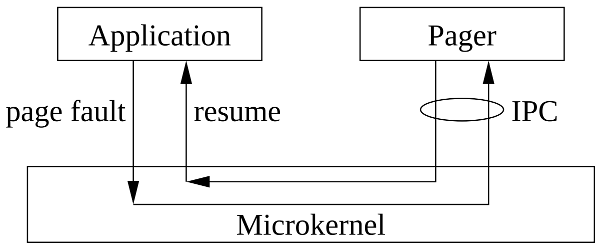

Figure 2.6: Page fault processing in L4.

2.3.3 Page Fault Handling

A thread can potentially access memory at any virtual address in its address space. The memory access succeeds only when a page with sufficient rights is mapped to the address. In all other cases the CPU generates a page fault and traps into the kernel. A monolithic kernel would now try to resolve the fault by mapping a page with sufficient access rights or by killing the thread due to an invalid access.

The L4 microkernel, however, provides only the mechanisms for address space construction. So the kernel itself cannot resolve the page fault, because address spaces are managed completely at user level with the operations map, grant and unmap. Therefore the kernel only provides a mechanism for reflecting page faults back to user level. This concept is called *external paging*.

In L4 every thread is associated with a *pager* thread. When the thread triggers a page fault it traps into the kernel. On behalf of the faulting thread the microkernel then generates an IPC to the pager. The message contains the fault address, the fault reason (read access or write access) and the thread's instruction pointer. The pager receives the message and analyzes the fault location and the fault reason. Now it can invoke appropriate fault handling, for example reading a block from disk (this can include IPCs to other subsystems, such as the disk driver), or copying the page contents into another page because the original page's policy is copy-on-write. Afterwards the pager sends the so prepared page back to the faulting thread using the map (or grant) operation. The microkernel inserts the mapped page into the page table and restarts the faulted thread. Page fault handling is illustrated in Figure 2.6.

The association of a thread to its pager is initially done at the time of thread creation. However, it can be changed at run time. It should be noted that the first thread of an address space is specially cased in L4V2: Its pager is specified during address space creation.

The concept of external pagers together with the operations map, grant and unmap allow for implementing arbitrary memory management strategies, for example pinned memory (that never gets unmapped), shared memory, copy-on-write or demand paging, at user level. All disallowed accesses are signalled to the pager.

Chapter 3

Design

Generalized Mapping, in short GM, is a new idea of the TU-Dresden's Operating Systems Group for a next-generation L4 microkernel. The main idea behind GM is to generalize the well-understood L4 mechanisms for memory management, the mapping and unmapping of pages, in order to use these mechanisms for all other kernel-provided resources as well. The resources provided by such an L4 kernel are memory pages, threads, tasks and communication channels.

As in current L4 kernels, a task is a protection domain. Thus a task can own resources of all the above types. Threads as the active entities can invoke operations on the task's different resources. A task has a task-local name space for each type of resource the L4 kernel provides. This allows a task to access the actual resource using local names without the need of a globally unique name[1]. However, when global names are needed, they can be implemented at user level. The local name space allows a task to manage names for resources with an arbitrary policy at user level. Furthermore, task-local names are also a step toward resource virtualization: A locally named resource can be backed by any physical resource of that type like it is already done with memory pages in current L4 versions.

A local name for a resource can be seen as a capability: When a task has a local name to a particular resource it can access or invoke operations on that resource. Without a name no operations can be invoked. So a local name, or a capability, allows certain operations on a resource. There are existing two types of operations: operations that can be used with all types of resources, and resource-specific operations.

The operations specific to a certain resource type can be invoked only on resources of that type. These operations are usually bound to the resource type's name space implicitly. So it is impossible to invoke these operations with the wrong type of resource. For example the read-memory operation always uses the virtual address space (the name space for memory) and therefore cannot be invoked on threads from the thread name space.

General operations can be used with all types of resources. However, when such an operation is invoked, it keeps track of the resource's name space. Thus name spaces cannot be merged using a general operation. Examples for operations valid on all resource types are map and unmap.

In a system with Generalized Mappings resources are transfered from one task to another using the map operation. This is basically a generalization of the mapping mechanism used for memory pages With this mechanism a task can grant access rights to any of its resources to another task. During the map operation the allowed access rights can be diminished. So the receiver of a mapping may have less rights than the sender. Widening the access rights with the map operation is not possible, because the kernel maps at most the sender's actual rights, even if it specifies more.

This thesis focuses on the IPC mechanism used in a Generalized Mappings kernel. So the following sections will discuss IPC and the IPC control mechanisms in detail.

[1]However, global names are needed for very special purposes e. g. physical addresses for DMA memory. So mechanisms have to be provided to translate local names to global names when needed. As current experience with memory shows, this can be done outside the kernel.

3.1 Requirements

In this section I will define the requirements the IPC mechanism of a kernel with Generalized Mappings should fulfill.

Trent Jaeger et al. [10] define the following six requirements for the system's trusted computing base (TCB). Since the IPC mechanism is part of the TCB, it has to fulfill these requirements.

- **Communication**: The system must be able to restrict the ability of a process to send an IPC to another process.
- **Authentication**: The system must identify the source of an IPC.
- **Authorization**: The system must be able to determine whether a particular operation on a particular object should be permitted.
- **Delegation**: The system must be able to control the delegation of permissions from one process to another.
- **Revocation**: The system must be able to revoke the ability of a process to communicate with another process or perform an operation on another process's objects.
- **Denial**: The system must be able to prevent denial-of-service attacks on processes.
- **Mechanism**: The system must be able to implement arbitrary access control mechanisms (e. g., optimized for the policy that they enforce) of its own choosing.

In addition to these six the following requirements came up in discussions before and during my work.

- **Local names**: All resources must be addressed using local instead of globally unique names. This allows changing of the actually used resource without changing its name. Wherever global names are needed, they should be created at user level.
- **Transparent thread structure**: The client should not need to have a priori knowledge about the thread structure of a server task. This allows the server to hide its internal structure or even change it transparently. With the new IPC mechanism one should be able to implement a wide variety of server structures — for example thread pools, dedicated threads per client, a server side distribution to worker threads, or single threaded servers — without having to add extra mechanisms for just that purpose such as auto propagation [21].
- **Isolation**: The provided mechanisms should allow to isolate subsystems from each other. Thus, communication restrictions must enforce that a task from one isolated subsystem is unable to communicate with tasks from other subsystems.
- **Synchronous IPC**: The IPC between communication partners should be synchronous to avoid complex buffering of messages inside the kernel. When asynchronous communication is needed, it should be implemented at user level.

3.2 Generalized Mappings for IPC Management

In microkernel-based systems the kernel provides the basic mechanisms for communication. Because it also provides the communication mechanism (IPC), the kernel must allow for restricting communication between tasks to enforce the desired security policy. The policy itself is not defined by the kernel, it is controlled by user-level policy servers but enforced by the kernel. Whenever the policy changes, the policy servers must be able to update the restrictions and thus permit new communication channels or revoke existing ones.

To achieve communication control, and thus communication restrictions, I introduce a new mechanism to the L4 microkernel: The right to send a message from one task to another task. This right is represented by a new kernel object called a *Task-Send Capability*, in short *send capability*. Possession of such a capability authorizes its owner to send messages using this capability. The send capability refers to the actual message receiver. Therfore these send capabilities establish a one-way communication channel[2]. When the receiver wants to reply, it also needs a send capability: a send capability referring back to the sender.

In this thesis the notation $\rightarrow B$ is used for a send capability. With the denoted send capability messages can be sent to task B. To name the owner of a send capability the notation $A \rightarrow B$ is used: Task A holds a send capability to task B.

In order to enforce security restrictions regarding the communication between tasks, the capability must be protected from modification by its owner. Otherwise a task could arbitrarily forge capabilities and thus send messages it is not allowed to send according to the system's security policy. The send capabilities used for GM IPC are kernel-provided objects. Therefore they are protected by the kernel. The owning task only gets a handle — the local name — to the send capability, the actual content is opaque to the user. This opaqueness allows for changing the capability transparently to the user because the user-visible name remains the same.

Send capabilities solve the authorization requirement: Only tasks possessing a send capability $\rightarrow B$ to a task B can send massages to task B. They are authorized to do so by a user-level policy server. Tasks without a send capability to task B cannot send messages to B.

Send capabilities can protect against some types of denial-of-service attacks. An attacking task without a send capability to a target task cannot send any messages to the latter and thus it cannot carry out DoS attacks. Tasks with sufficient send capabilities can still send useless "junk" messages preventing the target from doing useful work. This is not a problem, because after detection, the attacked task could ask the policy server to revoke the misused capability using the mechanisms described in the next sections.

Further on, send capabilities also solve the isolation requirement: Tasks inside an isolated subsystem receive only send capabilities to other tasks inside this subsystem. So these tasks cannot send messages to other tasks outside this subsystem. Similarly, no send capabilities to tasks from the subsystem are given to tasks outside the subsystem. Now only tasks inside the subsystem can send messages to other tasks within this subsystem. It is completely isolated from other communication.

3.2.1 Distribution of Send Capabilities

In the previous section I introduced the task-send capabilities. Now I describe the mechanisms used to propagate capabilities from one task to another. Afterwards I describe, how the well-known L4 mechanisms can be used to revoke the propagated capabilities.

A task in a system with Generalized Mappings holds some send capabilities. Each of these capabilities permits communication with another task. Over these established communication channels normal data messages can be sent, but also messages containing resources such as memory pages or capabilities. So a task can send its send capabilities to another task. Afterwards, both tasks own the same capability and can send messages using this capability. In the following this capability transfer will be called *forwarding* of a capability. Of course, a task can only forward capabilities itself owns. There is no way for the creation of new capabilities using the forward operation.

A completely new mechanism just for capability transfers is not necessary, it would even violate the minimalistic approach of L4. The L4 IPC system-call already provides a transfer operation for resources: the map operation. So far this operation is solely used for the mapping of virtual memory pages from one task to another. I use this operation also to transport capabilities from a sender task to a receiver task. Hence, the mapping of capabilities applies the same mechanisms as the well-known and well-understood mapping of pages from the task's virtual address space.

A slight extension of the L4 kernel has to be done to support a new type of flexpage, the *capability flexpage*. Such a capability flexpage consists of a base capability Id and the size of the flexpage, just like for normal memory flexpages. During the map operation all capabilities inside the given flexpage from the sender are inserted into the receiver's capability structures.

[2]IPC error codes can be used to leak information back from the receiver to the sender. To enforce strict one-way communication channels, a reference monitor has to be inserted into the channel (see Section 3.3.1).

Most of the classic capability-based systems do not keep track of the forwarded capabilities. Hence, these systems cannot easily revoke a capability, because they do not know which tasks have a copy of the desired capability (see also Section 2.2.3). Some systems, EROS [25] for example, provide an indirection level for the actual capability and can invalidate it there. EROS and Mach [1] provide capabilities with an use-at-most-once semantic, e. g. the EROS *return capability*. All copies of such a return capability become invalid when one of the copies is used. In Amoeba [29] capabilities are protected cryptographically (see Section 2.2.3) using a secret. Changing the secret invalidates all outstanding capabilities. Neither of these mechanisms is able to revoke a capability selectively from a subset of tasks.

The send capabilities used for GM IPC should avoid these problems. Again, a possible solution is already used in the L4 kernel for memory mappings. During mapping of memory pages the kernel keeps track which page from the source address space is mapped to which location in the destination address space. This information is kept in the mapping database. For each physical frame the mapping database records the map operations of a task and the virtual address in the destination space.

A similar database can be used to keep track of capability mappings. For each capability this database contains a tree[3] where the capability mappings are recorded. So the kernel can keep track of all mapping operations. Thus the kernel knows for every capability the tasks this capability is accessible from and also the location there. (Note: How a system is initialized and started is described in detail in Section 3.2.7. The total number of initial capabilities existing in a system is also describe there).

The mapping database stores the information how the task-send capabilities are distributed among tasks. This information can then be used to selectively revoke the capabilities a task directly or indirectly forwarded to other tasks. For memory L4 provides the unmap system call implementing this operation. The unmap operation traverses the mapping database for the requested frame starting from the current task. Then the page is removed from the address space of all tasks it was mapped to from the current task. Furthermore, unmap works recursively through all address spaces the original receiver has mapped the page to. This continues until the complete subtree of the mapping database is traversed and the pages are unmapped from all tasks inside that subtree.

With the mapping database for capabilities and the aforementioned capability flexpages the unmap operation can be extended to revoke forwarded capabilities. Hence, we also have an effective way for the selective revocation of send capabilities.

The unmap operation is safe, because a task can only unmap capabilities it owns. Furthermore, the receiver of a capability mapping implicitly agrees to a possible unmap when it accepts the mapping. for this reason no further authorization or agreement is necessary for the unmap call.

With the introduced mechanisms and operations the rights-delegation and rights-revocation requirements can be solved. The map operation during IPC is used to delegate the rights from one task to another. The unmap operation together with the mapping database is used for the revocation of previously granted communication rights.

3.2.2 Preventing the Unwanted Distribution of Rights

As described in the previous section, task-send capabilities are distributed between tasks by mapping them from one task to another. Up to now, there are no restrictions whether a task can forward its send capabilities (or other resources such as memory pages) or not. However, when the mapping operation is allowed without any restrictions, a new problem arises: The unwanted distribution of rights.

For an illustration look at Figure 3.1. A subsystem with three tasks and a well defined information flow (send capabilities) is shown. For example, this subsystem could be a part of a video player: X is a controlling task getting user input (e. g., seek). It instructs the disk driver Y to read blocks from disk. The blocks are transfered to a task Z that decrypts them for presentation (using a trusted screen driver not shown in the figure). The defined data flow assures that no decrypted data can leak back to the disk.

However, when allowing unlimited capability mappings, X can map its capability $X \rightarrow Y$ to Y. Y in turn maps this capability to Z establishing an unauthorized backward channel. This new channel is

[3] The kernel prevents cyclic mappings and allows a capability in a task to origin from at most one task (the mapping task). Therfore the mapping-database structure is a tree.

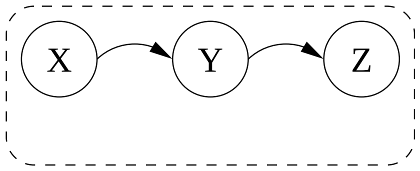

Figure 3.1: A subsystem with a defined information flow.

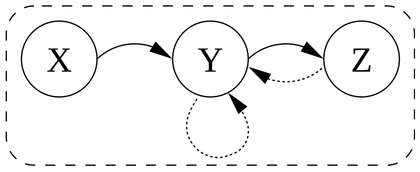

Figure 3.2: The same subsystem with unwanted communication channels.

unwanted, because it allows Z to send the decrypted video data back to the disk driver. The new situation is shown in Figure 3.2, the dotted arrows are the unwanted communication channels derived from $X \to Y$.

For discussing solutions for this problem look at the scenario from Figure 3.3. Given are two groups $G1$ and $G2$ of untrusted tasks. Inside a group the tasks have send capabilities so they can communicate with each other. Every group has a trusted member ($T1$ and $T2$), which can additionally communicate with tasks outside its group. In the Clans & Chiefs model [14] these tasks would be the chiefs. Because all communication crossing a clan boundary is redirected to the chief, the unwanted rights distribution problem does not exist here: The chief intercepts the message and can discard disallowed or unwanted parts before forwarding the message to the destination. In practice, however, this very restrictive communication has turned out to be inefficient, because for one message sent from B to K, at least three IPC operations have to be invoked: B to $T1$, $T1$ to $T2$ and $T2$ to K.

In a system with capabilities the trusted tasks could establish a direct communication channel between dedicated members of $G1$ (B) and dedicated members of $G2$ (K) for fast inter-group communication. Hence task B gets a send capability to task K. Forwarding this capability inside $G1$ is unproblematic because B could also forward messages from other tasks inside its group acting as a proxy. The problem here is that the untrusted task B can forward the send capability it has to task C ($B \to C$) over the established channel to K. This forwarding will establish a new communication channel between the groups (dotted arrow in Figure 3.3), circumventing the trusted tasks, which are unaware of this new communication channel from K to C. The problem here is that the newly created channel cannot be closed easily by the trusted tasks: Revoking the send capability $B \to K$ from B does not revoke $K \to C$. Things get even worse, if other resources such as memory mappings are transfered, establishing a shared memory region between the different groups which is a high bandwidth channel.

A possible solution for this problem could be to restrict whether a send capability can be forwarded to another task or not. In the given scenario all send capabilities B receives would be tagged as nonforwardable. Therefore no new direct communication channel (e. g., from K to C) can be established. B, however, can still map memory pages to K and thus establish the shared memory channel. So this forwardable flag has to be implemented for every type of resource in the system to prevent unauthorized channels. This means that in the scenario all resources (memory pages, send capabilities and possibly others) of group $G1$'s members have to be tagged nonforwardable, otherwise a group member, for example C, can forward its mappings to B without tagging, and B in turn can send that mapping to K establishing a new channel.

Instead of limiting the forward-ability of send capabilities (and other resources too), the communication channels could be restricted to allow mapping operations or not. For the above scenario this means that the send capability $B \to K$ is tagged to disallow the transfer of mappings. Now B is unable to establish new communication channels between K and tasks inside $G1$, because it can only send plain data messages but no mappings to K. All other communication channels inside $G1$ can be tagged to allow mappings without

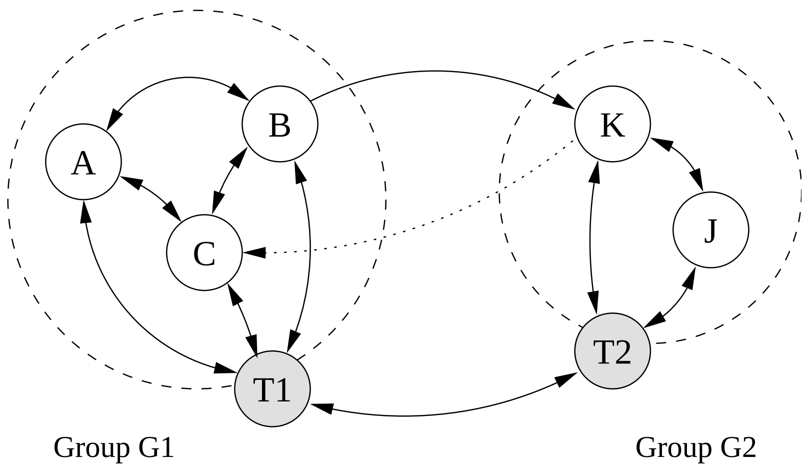

Figure 3.3: The unwanted distribution of rights.

implications on security because communication from and to the outside is restricted ($B \to K$) or filtered by $T1$.

The tagging of communication channels introduces no new mechanism, but is a permission on the communication channel.

3.2.3 Capability Fault Handling

Possession of send capability authorizes communication to a particular task. The destination task is determined and the capability is validated during the invocation of the IPC operation. Without a proper capability the task cannot send the message. So a task may try to invoke an IPC operation using an invalid send capability (e. g., a capability it does not own). The invocation of IPC is forbidden with this capability and a fault occurs. This fault that has to be handled by the GM-IPC mechanism.

There are four different causes of faults during the validity check of the used capability. First, the given local name of the capability can be out of the range supported by a particular GM implementation. Second, the capability may be invalid because it is not mapped. Third, the capability may be used for an operation it is not intended for. Fourth, the capability me be lacking a required permission. The third case cannot happen with the send capabilities from this thesis because the task-send capabilities can only be used for IPC. We handle the fourth case by silently discarding mapping requests if the transfer of mappings is not allowed with the used capability.

For the first type of failure, the capability is out of supported range, the IPC operation should abort and return the error code *invalid destination*. Application programmers must pay attention to this condition and handle it accordingly.

Failures of the second class occur, when a task tries to use a send capability, which is not mapped to that task. This can happen either by using an arbitrary capability the task never possessed, or by using a capability that already got unmapped. From the kernels point of view both possibilities are the result of invoking IPC with an invalid capability. This resulting fault is called a *send-capability fault*.

In classic L4 only memory access faults, the page faults, can occur. The kernel translates the page fault into an IPC and sends it to the faulting thread's pager. A similar mechanism can be used for handling capability faults.

For handling capability faults, every thread in a GM system has a capability pager. On a capability fault the kernel generates capability-fault IPC and send it to the capability pager. Like a page-fault IPC, the capability-fault IPC consists of the faulting capability name and the instruction pointer where the fault occurred. The capability pager inspects the message and extracts the faulting capability name. Then the pager resolves the fault by mapping an appropriate capability to the faulting task. Afterwards the faulting operation is restarted, This type of error handling is transparent to the faulting thread, such as page faults are today. It is useful to fulfill parts of the transparency requirement: A pager can revoke capabilities from its clients at any time. During the next IPC operation a fault is generated, and the pager can send another capability back to the client. Without further interaction, the client now sends to another task as before the replacement. This mechanism can be used when server tasks need to be restarted, when splitting a large server into smaller ones, or when joining several server tasks into one task. Moreover, in Section 3.3.1 I show in detail, how a communication interceptor (e. g., a debugger) can be inserted into an existing communication channel.

However, with the described revoke and fault-in mechanism one might not be able to build systems with arbitrary capability distribution between tasks. For example, a system is built, where a client sends a reply capability together with its request to a server. Before the server replies, the client revokes that capability. In the scheme above, there is no chance for the server to detect that the capability is lacking, hence the server faults. To eliminate this fault, the server must detect the missing capability. The only reliable check for the validity of the capability can be done during the IPC operation. But instead of generating a fault IPC and restart the faulting IPC afterwards, the IPC operation returns an error code. This error code can be used by the server to do the appropriate error handling.

For the discussed reasons, a system implementing the Generalized Mappings IPC mechanism should provide both types of error handling, transparent fault IPC with restart and IPC abort with error code.

3.2.4 Local Id's on the Sender Side

In the preceding sections I introduced the task-send capabilities and their handling in general. In this section and in the following sections I describe how these send capabilities are actually used for the communication between threads.

To use the send capabilities a sender thread needs to name the capability it wants to use for sending the message. Since a task, and thus the threads within, can own many different send capabilities, the sending thread must be able to distinguish between the different send capabilities using different names. Hence, every task requires a name space for naming the send capabilities: the *send-capability name space*. Like the name space for memory, the virtual address space, the send-capability space is local per task. This enables a task to manage the capability name space on its own, applying any policy for the send-capability naming it wants.

The send-capability space is indexed by natural numbers. Every number represents a slot for one send capability. A slot is either empty, no capability is mapped there, or it contains a valid send capability. In the following the term *capability Id* is used for the slot number. For a sender, the capability Id represents a send capability, and thus a communication channel. Since every task has its own send-capability name space, the capability Id's are also local per task. Different tasks can have different send capabilities at the same capability Id. The capability Id can be interpreted (and of course used) also as the senders local name of the intended receiver of messages sent using this capability Id.

The send-capability map and unmap operations described in Section 3.2.1 work within the capability space. Both operations use capability flexpages consisting of a base capability Id, and the size of the flexpage. As the base address in memory flexpages the base capability Id is aligned according to the flexpage size.

The send-capability name space also solves some of the defined requirements. It allows a task to use arbitrary local names for their communication partners. Further, together with map and unmap operation the send-capability space provides the transparency of the actual message receiver: The sender's pager can revoke a send capability at any time. Later, when resolving the according capability fault, the pager can forward another send capability to the sender. The sender does not notice this change of the receiver, because the sender's local name remains the same. Refer to Section 3.3.1 for an example.

3.2.5 Local Id's on the Receiver Side

Task-send capabilities authorize the communication from a sender to a receiver as described. Consequently, when a task receives a message, it knows that the sender was allowed by an authority to send the message. The problem now is, how the receiver can identify the source of the received message. Or, in other words, how it can figure out, which of all possible senders actually sent the message.

This sender identification provided to the receiver must be unforgeable by the sender, otherwise a malicious sender can spoof it and send messages pretending to be another task. That's why the reliability of the identification must be enforced. In Generalized Mappings the kernel is a trusted entity, therefore the kernel provides the mechanisms for enforcing an unforgeable identification.

A naive approach would be the transmission of the sender's thread Id like in current L4. This solves the identification problem and can also be useful for accounting purposes. But it violates the transparency requirement and the idea of Generalized Mappings, where global Id's should be omitted if they are not needed for very special purposes (e. g., DMA memory). Therefore another mechanism is needed for sender identification.

The solution I propose for GM IPC is that every message gets an unforgeable identifier stored inside the capability used to send the message. Since the capabilities are kernel-protected objects, the sender cannot forge this identifier. This identifier is called the *principal Id*, in short *PID*. I do no call it sender Id because this would imply a particular use. The PID can be used for other purposes as described below.

For the kernel the PID is an opaque bit string of length n . The bits do not have a special meaning for the kernel. On IPC these bits are inserted into the message by overwriting the first n bits of the message. So the first part from a received message, n bits, is enforced through the kernel and unforgeable by the sender.

The receiver can interpret the PID in any manner. For example, a receiver uses the PID only for identifying the sender, thus the PID is a sender Id. When replying to the sender, the receiver maps the received PID to one of its send capabilities. More complex receivers can use the PID as object identifiers (file descriptors for example) the message content has to be applied to. Since the PID is unforgeable, the receiver can trust the object Id without further checks. Another possibility is the encoding of the function Id the receiver calls with the message. For more scenarios and examples refer to Section 3.3.

The PID solves the authentication requirement, because it is unforgeable inserted into every message. It also gives the receiver a local name space for its requesters. The remaining questions now are, how the principal Id's are managed, who manages them, and how they are placed into the send capabilities.

PID Management

The principal Id comes from the send capability. So the actually used PID has to be inserted into the capability. Send capabilities are forwarded from one task to another using the map operation. During this map operation the forwarding task specifies the PID and its length. These values are inserted into the forwarded capability.

But this PID insertion mechanism must prevent a task from forwarding send capabilities with an arbitrary PID. Otherwise a malicious task can insert any PID when forwarding a send capability, and thus rendering the PID useless for any kind of identification. Therefore only new bits can be appended to the PID. The already existing PID bits from the original send capability are inserted into the newly specified PID. The new PID length is the maximum of the PID length of the existing capability and the newly specified length during capability forwarding.

So the pagers in a pager hierarchy can build a hierarchic PID space by specifying some more bits of the PID when forwarding a send capability to the next pager.

Acting on Behalf

In classic L4 with Clans & Chiefs the Chief of a Clan can send messages to tasks outside that Clan pretending to be sent by any task inside this Clan. This is necessary, as every message a task inside the Clan sends to the outside is redirected to the Chief. When the Chief then forwards the message to the intended destination, it acts on behalf of the original sender.

With the send capabilities and the principal Id GM IPC supports also tasks acting on behalf of other tasks. Looking at the example from Figure 3.4, the pager P owns a send capability $P \rightarrow S$ with PID pid_p. The pager established a channel $C_1 \rightarrow S$ by forwarding its send capability $P \rightarrow S$ specifying the PID pid_c. Later on, P also forwards this capability to C_2 also with the same PID pid_c. Since C_2 now owns a send capability $C_2 \rightarrow S$ with the same PID pid_c as C_1, it can send messages on behalf of C_1. The server S does not notice the different sender.

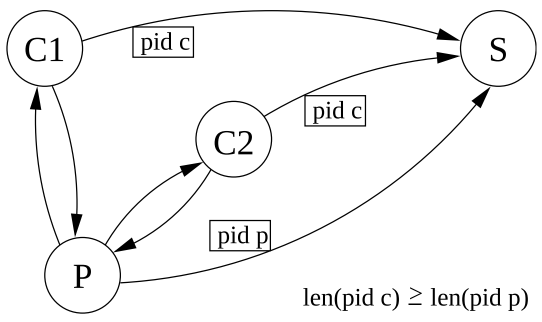

Figure 3.4: Acting on Behalf.

Also a pager can always act on behalf of its clients, because it is the one who forwarded the capabilities to the clients. If the pager forwards the capabilities unchanged, that means without further restricting the PID, the scenario is similar to the previous one with $pid_p = pid_c$. When the pager forwards the capability with additional PID restrictions, hence the pager specifies more bits for pid_c than pid_p has, the pager itself has to put the additional bits it specified for pid_c into the message because only the shorter pid_p is inserted from the kernel. Also here the server does not notice a difference in the received messages from the pager or the client as both start with the same bit pattern, the client's PID. Thus, the precondition for impersonation is that the PID of the impersonator is a prefix of the PID of the impersonated task.

In Section 3.3.1 demonstrates how acting on behalf is used when intercepting messages.

3.2.6 Receiver Selection

With the Generalized Mappings IPC mechanism messages are sent to a task specified by the senders send capability. While this allows to hide the receiver's thread structure from the sender, the task addressing has the problem that threads, and not tasks, are the active entities inside a system. Only threads can invoke the IPC operation for sending or receiving messages. So the thread actually receiving an incoming message to the task has to be selected by some mechanism.

Usually not all threads inside a task are equal in terms of message reception, because the application might associate some given semantics with the threads. Another problem is that all threads would have to provide message buffers for all possible incoming messages. More problems may appear when threads have different priorities in order to prioritize some clients or operations.

The solution GM IPC offers for this thread selection problem is the specification of a *message criteria*. A thread specifies the message criteria for messages it is willing to receive. The message criteria consists of a bit string and a length n . For an incoming message the kernel matches the first n bits of the message with the bit string from the message criteria. When the massage matches the criteria, the message is delivered to the waiting thread. Otherwise the message is incompatible with the specified message criteria and not delivered to that thread.

This type of matching is called *prefix matching*. Prefix matching was chosen for receiver selection in GM IPC because in current systems the first message part often contains an object Id or an operation Id to which the remainder of the message has to be applied to. With prefix matching a receiver thread can wait for specific object or operation Id's.

This content-based receiver selection can be used to implement a wide variety of different selection policies at user level. The three most common cases, open wait, group wait and closed wait, I will describe now.

For an open wait a receiver thread specifies an empty message criteria with length 0. This pattern matches every message. Therefore the thread is a candidate for receiving any incoming message, as every message is compatible.

For a closed wait, L4 speech for waiting on exactly one communication partner, the receiver thread specifies a bit string and its length as message criteria. Additionally it must be guaranteed that only threads inside one sender task can send messages which match the given criteria. This constraint can be assured with a proper PID distribution: The first part of the message is used for prefix matching with the message criteria. On sender side, the first message part is also the part that is replaced with the principal Id from the capability. So the PID is one part of the message which matched with the message criteria. When the receiver can trust the sender's pager, i. e. the pager does not map the send capability with the same PID to other tasks, a closed wait for one sender can be assured.

The third common case is called group wait. Here the receiver thread also specifies a bit string and length as message criteria. Although, in contrast to closed wait, the specified criteria has a shorter length, and thus suits to several incoming messages. Again, with proper PID distribution group wait can be used to wait for messages coming from a group of other tasks, but not from tasks outside that task group. Another example are threads of receiver tasks, which use the first message part as an object Id. Group wait can be used for waiting on messages regarding to any object out of group of objects. For an example refer to Section 3.3.2.

Together with the described hierarchical PID distribution, the content-based receiver selection provides protection from some types of denial-of-service attacks. All messages of an attacking sender containing its PID. Thus, this sender can carry out a DoS attack only against receiver threads waiting for messages with the senders PID or a shorter subset thereof. Of course, as today too, open-waiting threads are also affected by a DoS attack. All other threads are not affected by the attacking sender.

Considerations for Multiple Receiver Threads

As described, threads waiting with a specific message criteria. The kernel chooses the actual receiver by comparing the first bits of the incoming message with the message criteria of each receiving thread. For multi-threaded receivers it is possible that the criteria of more than one thread matches the incoming message. This happens either when several threads are waiting with the same message criteria or when one waiting thread specifies a prefix of another thread's criteria.

In both cases the kernel has to choose one out of all possible receivers. In the first case, several threads with the same message criteria, all possible threads are equal from the kernel's point of view. Choosing the last running thread for receiving the message is a good idea here, because the working set of this thread may still reside in the cache. Thus this choice can outperform the selection of another thread.

For the second case, choosing the first fitting thread, or even a random one, is also not a good idea, because with such a strategy there is absolutely no control to which thread the message is actually delivered. A more deterministic way is that the thread with the most specific message criteria gets the message. The idea thereby is that this thread might have more a priori knowledge about the message than the other threads. Thus it can benefit from the kernels choice. Another point is that threads with a shorter message criteria accept a wider range of messages. Choosing a thread with a longer message criteria does not block the reception of other messages out of this range. The more specific receiver cannot receive these messages due to its longer message criteria.

This afore described selection mechanism can be used to implement thread pools. Many threads are waiting with the very same criteria. An incoming message with this content is delivered to one waiting thread out of the pool using LIFO selection. No further special mechanism, such as auto propagation [30], is needed for just that purpose. Answering back to the sender is not an issue, because the needed send capability is owned by the task. Each thread can use it for replying.

Another example are worker threads. Here, only one thread receives messages from task-external sources. Then it analyzes the message content and forwards the message to another local thread that does the actual work. This allows a task to implement arbitrary message distribution at user level based on the content. For the same reason as above, replying to the original sender is also not a problem here.

3.2.7 Starting a System

I explained the motivation and usage of the send capabilities. Now I discuss, how a system based on Generalized-Mappings IPC starts up. Afterwards I show the effects on task creation and start.

When a system starts, the hardware is checked and initialized to a known state. Afterward the boot loader loads the kernel and possibly a couple of initial tasks into memory. Then the kernel is started, initializes itself, and takes over the control of the system. A classic L4 kernel then creates a root address space called σ_0 that represents the physical memory. This address space has an idempotent mapping of virtual addresses to physical addresses. All other address spaces are created empty. Further on the kernel creates two tasks: Sigma0 and the root task. Sigma0 has the address space σ_0, therefore it is the root for recursive address space construction. The root task starts up the rest of the system by loading and starting the needed tasks.

This strategy can be extended for a system with Generalized Mapping IPC. The kernel has to create an initial send-capability name space cap_0 with an idempotent mapping of send-capability Id's to tasks. The idempotent mapping allows for easy conversion between task Id's and the corresponding send-capability Id's. No additional mechanism is needed for it. For every possible task in a system there exists exactly one send capability in cap_0. All send capabilities in cap_0 have an empty, zero-length PID. Therefore a task having cap_0 as its send capability space can send any message to any task, because nothing inside the message is overwritten. So this task can also impersonate all other tasks.

The question now is, which task gets the initial space cap_0. Basically there are two possibilities. First, to create an additional task *Capability0* that has cap_0 as its send-capability space. Second, cap_0 goes to Sigma0.

In the first case, the additional task Capability0 owns all send capabilities to all tasks, hence it is the root for capability distribution, while Sigma0 owns all the memory pages. However, this variant has a chicken-egg problem: The virtual address space of Capability0 is empty, therefore a page fault is raised immediately after the start. Unfortunately, Sigma0 cannot send the required memory pages, because its send-capability space is empty. A solution for preventing such a deadlock during startup is that the kernel instantiates some initial capability and memory mappings between Sigma0 and Capability0. However, when more types of resource mappings, such as task mappings or thread mappings, come into play in the future, these problems get even worse.

These problems do not exist for the second variant. Here both name spaces, σ_0 and cap_0, are owned by Sigma0. So Sigma0 is the root for both recursive address space construction and recursive send-capability space construction. By the latter, Sigma0 can also partition the PID space each next-level pager manages.

The second variant is the more elegant way, because it does not add additional complexity to the kernel. Everything regarding a policy can be done at user level. Sigma0 can provide and enforce the systems basic security policy by partitioning the PID space.

Creating a new Task

During system start, but also afterwards, new tasks are started. Initially, the virtual address space and also the send-capability space of a newly created task are empty. The empty send-capability space is problematic, because the task does not have a send capability to its pager. Therefore the task, more precisely the kernel on behalf, cannot send fault messages to the pager. So fault handling would be impossible.

In classic L4 this problem is solved by specifying the initial pager on the task-creation system call. So the newly created task can (and of course will, because it does not have any code pages mapped yet) raise page faults that are handled by the pager.

A similar strategy can be used for creating tasks in a system with the Generalized-Mappings IPC mechanism. Here the send capability to the pager has to be specified for the task-creation system call. The creating task specifies one of its send capabilities, which then is mapped to the newly created task and used there for sending fault messages. So the new task can raise faults, and these faults are resolved by the pager.

3.3 Scenarios

3.3.1 A Message Interceptor

In this section I demonstrate how the mechanisms of Generalized Mappings can be used for the insertion of an intercepting task into an existing communication cannel. Afterwards all messages sent through this channel can be inspected by the newly inserted task. Thereby, the interceptor insertion can be transparent

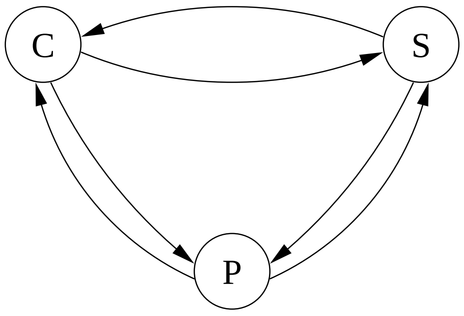

Figure 3.5: Three tasks, client can communicate directly to the server.

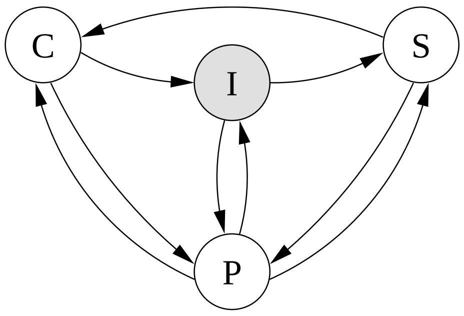

Figure 3.6: After inserting the message interceptor.

to all other involved tasks[4]. The interception task, for example, can be a debugger, an IPC tracer, or a filter (e. g., a reference monitor) that drops unwanted messages.

Message interception is illustrated with the scenario from Figure 3.5: The system is running and there are three tasks: a client C, a server S and their pager P (in principle S can also have its own pager, but this would make the scenario slightly more complex). During the startup of C a communication channel from C to the pager is established, and thus C can send fault messages to P. The pager also established a channel from itself to C, so that it can reply to the fault messages. The same holds for the server. As the client wants to send requests to the server and the server answers them, the pager established channels from C to S and from S to C by mapping the appropriate capabilities.

At some point in time the pager decides to observe the messages the client sends to the server. Therefore the pager unmaps the send capability $C \rightarrow S$ from C. So C can no longer send messages to S. Upon the next request C tries to send, the kernel generates a capability-fault IPC and delivers it to the pager. The pager starts an interposing task I. I gets a send capability to P so that P can act as its pager. Obviously the pager has also the right for sending messages to I.

After the creation of I, the pager replies to C's fault message with a send capability mapping to task I. From its point of view, C can now communicate again with S, as only the capability internal data has changed, but C's local name for S remains the same. The intercepting task I can now examine the incoming message. It can apply any policy to the message including changing the content or even drop it. When I decides to forward the (potentially modified) message to S the first time the kernel generates a capability-fault IPC to the pager. The pager resolves this fault by mapping a send capability $\rightarrow S$ to the interceptor, applying the same or a prefix of the principal Id it formerly mapped to C. The new situation is shown in Figure 3.6.

From S's point of view it still gets messages from C without any notice of the intermediate task I, because I is able to send the same messages as C could — the authentic principal Id of I is a prefix of the PID of C. Therefore the insertion of the interceptor can be transparent to both C and S.

[4] When a task installs a local fault handler it can detect the insertion. Also with timing-based attacks the insertion can be detected.

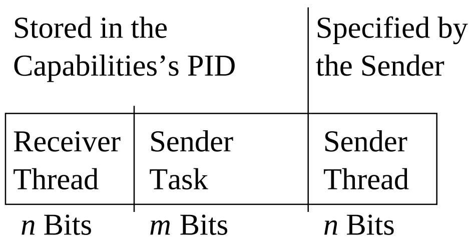

Figure 3.7: An encoding for global Id's.

When message inspection on the way back from S to C is demanded, the previous steps also have to be applied to S. The pager then might decide to use the already existing I instead of creating a new one. In the latter case I can inspect both messages from C to S and the replies from S to C.

Removal of I from the communication channels can be done in the same way as the insertion: The pager unmaps the send capability $C \rightarrow I$. On C's next IPC invocation a fault message is generated and the pager replies the original send capability sending to S.

One problem when inserting an intercepting task is that IPC between C and S is no longer synchronous. C sees a successfully sent message even when the IPC between I and S results in an error or, for example, S has been destroyed meanwhile. This is a general problem when using message interception. Trent Jaeger et al. proposed a solution in [11] for classic L4. The mechanisms they introduce are keeping the source of a message blocked when the message is intercepted until the final receiver is reached, and sending back a notification about IPC success to unblock the sender. These mechanisms are orthogonal to the GM-IPC mechanism, they can be implemented in a similar way for GM-IPC. However, additional send-capabilities might be needed for the unblock messages.

3.3.2 Global Id's

In the L4 versions L4V2 [18] and L4X0 [30] the kernel provides a fixed scheme for Id management with globally unique thread Id's. Such an Id consists of a task Id, a thread Id, and a generation number for uniqueness in time[5]. This section describes, how to emulate this policy on top of a system with Generalized Mappings IPC.

The global Id's are used for thread identification. For IPC the sender of a message specifies the global Id of the receiver. The receiver gets the senders Id. Because these Id's are global, unique, and unforgeable, the receiver can rely on it[6]. Another benefit of global Id's is that a thread has always the same Id when communicating to arbitrary other threads. That's why, in systems with global Id's the thread Id's are often used for authentication and identification purposes in applications.

When emulating global Id's with GM the aforementioned conditions must hold. A global thread Id consists of `m` bits task Id and `n` bits thread Id. Therefore a task can have up to 2^n threads. On the sender side, the capability Id can be used directly as the global Id of the destination thread. The first `n` send capabilities refer to threads of task `0` , the next `n` to task `1` and so on. For receiving the sender's global Id on the receiver side, a clever encoding of the global Id for the first message word and therefore the PID is needed. Additionally, this encoding must guarantee that the correct receiver thread is selected for message reception. The encoding must also work for multi-threaded senders.

Figure 3.7 shows an encoding for the first message word, which accomplishes both requirements. The first part, *receiver thread*, is used on receiver side for selecting the right thread. For an L4-style open wait, a receiver thread does a group wait specifying its thread Id as message content. The receiver task Id is not needed here, because it is only relevant on the sender side – the used send capability there refers to the right receiver task. The next two fields contain the senders task Id and thread Id. Closed wait for one sender thread is implemented by specifying the complete triple on a group-wait operation.

[5]Additional fields of the different L4 versions, such as clan Id or site Id, are omitted here. They can be seen as part of the task Id.
[6] Deceiving by chiefs is not considered as a problem, for discussion see [14].

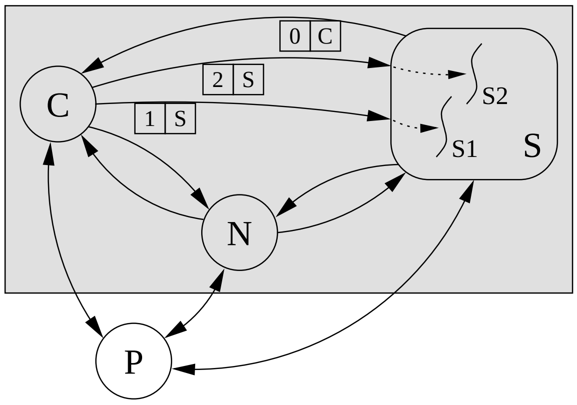

Figure 3.8: An example for tasks using global Id's for communication.

The fields *receiver thread* and *sender task* are put into the PID of a send capability. The latter is necessary to prevent malicious tasks from forging their Id. The first one is needed due to the afore described wait semantics together with the fact that the PID is always a complete bit string without holes. The *sender thread* field is specified by the sender thread. This is safe, because a task is a protection domain. Putting *sender thread* into the PID too would require much more send capabilities inside a task (one capability for each possible combination of sender thread and receiver thread, which grows quadratic with the number of tasks) while giving no additional security, because a thread can always use the send capabilities intended for another thread of the same task. Thus, in both cases it can impersonate any other task-local thread.

An example scenario is shown in Figure 3.8. There are a client C, a multi-threaded server S and a name server N. These three tasks using global Id's for communication with each other. The threads S_1 and S_2 of S registered themselves at the name server for the services they provide. The client (single-threaded, thread C_0) wants to use a service provided by S_1, so it first asks the name server for the Id of the thread providing it. The replied thread Id, the thread S_1, is used as destination by the client. The pager P manages the send capabilities of the tasks.

When a thread uses a send capability the first time, a capability-fault IPC is sent to the pager. The pager extracts the faulting capability Id from the fault message and calculates the requested task and thread out of it. Then it resolves the fault by mapping a send capability referring to the task with the calculated global task Id. Thereby the PID fields for *receiver thread* and *sender task* of this capability are set to the calculated receiver thread and the faulting tasks global task Id.

In the example C faults on capability $S_1 = S * n + 1$. When resolving the fault the pager P maps its send capability $P \to S$ with the PID $\boxed{1}\boxed{C}$ to C. Now C can send messages to S_1. After handling C's request, S_1 uses the received global Id for replying. Here also a capability fault occurs. The pager resolves that fault by mapping its $P \to C$ to S with the PID $\boxed{0}\boxed{S}$

I showed, how the current situation of global Id's can be emulated with GM IPC. However, the emulation has some limitations. The pager has unlimited send capabilities to all tasks that use global Id's, so it could impersonate all that tasks. This is a non problem, as a task must trust its pager anyway. Another limitation is that a communication attempt with a not existing thread of a task does not result in an error code. Because the task exists, the pager maps a capability, but no thread is waiting for the *receiver thread* specified in the PID. Two more things can be seen as limitation, but also as benefit: a thread can impersonate other threads of the same task, and a thread can wait for messages intended for other threads by specifying a wrong Id when waiting for a message. Both cases cannot be prevented, because the protection domain is always a task. However, they need not be prevented from a security perspective. The first case is useful for request propagation to worker threads. The actual worker thread replies directly to the client using the propagator's Id as sender thread Id. The second case allows the implementation of thread pools.

3.4 The Prototype

In the previous sections I described the goals and the general design of the Generalized Mappings IPC mechanism. Now I am going to describe the architecture of the prototypic GM-IPC implementation I developed during this diploma thesis. The concrete implementation challenges are discussed in the next chapter.

3.4.1 General Architecture

At the beginning of my work was clear that during the available time for a thesis it is impossible to work myself into the depths of Fiasco and incorporate a completely new IPC mechanism. Also it was unclear, how GM-IPC exactly works and what are the effects on the remaining system components.

So I decided to implement the GM-IPC mechanisms inside a library at user level on top of the existing L4V2 infrastructure [18]. Programs that use the Generalized Mappings mechanisms are linked against that library. They use the library provided functions for communication instead of the kernel provided ones. Furthermore, the library provides send-capability management, mapping and unmapping of send capabilities and capability-fault handling.

The implementation as user level library has several benefits: I don't have to define a new kernel binary interface (ABI) for capabilities and the receivers message specification. The library's interface can consist of ordinary function calls. In contrast to a binary system-call interface to the kernel the parameters of these function calls can be type checked at compile time. This eases debugging and prevents mistakes when functions or data types are changed in the implementation, but not in the corresponding header files. Such checks are not possible with the current Fiasco kernel, because the kernel does not use user-level headers and vice versa.

Another benefit from a library is the more rapid development using the existing infrastructure. So new functionality can be added step by step. Inside the kernel, large parts of the IPC path have to work reliable (think about memory mapping) before it can actually be used and tested by user programs. In addition, user level code is often easier to debug because some types of error, such as invalid accesses, are trapped by the kernel. Also debugging software can be used. However, the ability to debug should not be overemphasized as Fiasco has a powerful in-kernel debugger for both user level and kernel level debugging.

Besides the benefits an implementation as library has also limitations. Communication control is relevant for system security and also for denial-of-service prevention (see Section 3.2). When implementing send-capabilities purely at user level the security requirements cannot be met, because a hostile application can access and even modify all the library's data structures. In an in-kernel implementation these security-relevant data structures are protected by the kernel. Accessing them from user level will be detected and denied by the kernel's fault handler.

These important limitations, however, do not interference with the main goals of this thesis, which are to discuss and implement GM-IPC and see whether it is useful and should be implemented inside a next-generation L4 kernel.

The developed library defines the necessary data types and functions for library internal and user visible send-capability handling. Functions for sending and receiving messages using GM-IPC are also provided. Task and thread management are not subject of the developed library. For those the mechanisms provided by the native L4 kernel have to be used.

Chapter 4

Implementation

In the previous chapter I focused on the fundamental design of Generalized-Mapping IPC. I also explained the reasons for implementing the prototype as a library at user level and the basic library design: The library provides the Generalized Mappings IPC functionality. Also the send-capability management and the capability mapping is provided.

This chapter points out the concrete implementation of GM IPC inside the developed library. In the following section I present the library's architecture and the provided data structures. Thereafter I present the Generalized Mappings IPC path in detail. Finally, in Sections 4.4, I discuss the matching requirements and possible algorithms for receiver selection.

4.1 Library Architecture

I implemented GM IPC prototypically as a library at user level. All application using the Generalized Mapping mechanisms are linked against the library. This is shown in Figure 4.1

The most important part of the library is the distributor thread. This thread is started at program startup when the library is initialized. The distributor thread receives incoming GM-IPC messages and selects the actual receiver thread from all waiting threads. I chose a design with an extra thread, because in GM-IPC messages are sent to tasks using send capabilities. Furthermore, the distributor thread is also responsible for mapping and unmapping of capabilities.

The distributor thread manages the library's data structures; the lists for sender threads, receiver threads and timeouts. It also manages the mapping database for capability mappings.

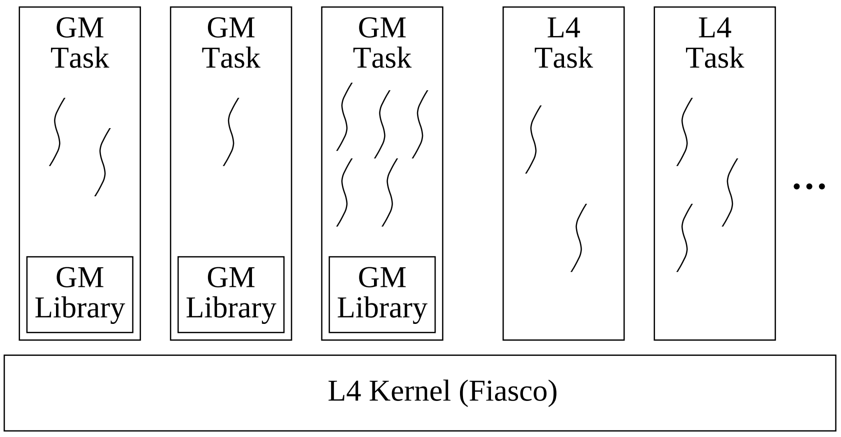

Figure 4.1: A system with GM tasks and non-GM tasks.

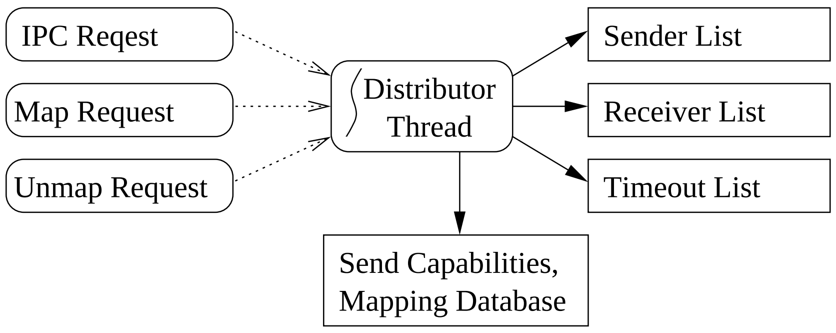

Figure 4.2: The distributor thread.

The distributor thread, the requests to it and the important data structures are shown in Figure 4.2.

4.2 IPC Implementation in Detail

The main objective for this thesis was the development and test of the Generalized Mappings IPC mechanism. As the library works on top of an L4V2 kernel, I decided to extend the existing data format and the function syntax of V2-IPC slightly for using send capabilities and the new receiver-selection mechanism.

The new GM-IPC send function takes a capability Id instead of the receiver's thread Id. All other parameters, the IPC descriptor, the two "register" words and the timeout are the same as in the current send operation for L4V2. So these parameters can be given directly to the underlying L4V2 IPC system call. The capability Id is not the only new field: An additional first message word, to which the PID is applied, is needed. The PID cannot be applied to the first "register" word, because this word is interpreted during memory map operations by the underlying L4 kernel. Therefore the message should not be changed by my library.

The GM-IPC receive function takes a structure for the message criteria. The additional first word from the send operation is matched to the message criteria. When the word matches it is delivered back to the receiver in the message criteria structure. All other parameters for the receive operation, except the now superfluous sender thread Id, are the same as in L4V2.

From the users point of view the only changes are the name of the IPC functions and the first parameter, which is no longer a thread Id. All other fields and also the message format remains the same. This also holds for a combined send-and-receive operation.

My library capsulates the GM IPC path. When the send operation is invoked, the used capability tested for validity. In case the capability is invalid (i. e., not mapped) fault handling is done. Otherwise the capability's PID is applied to the first word. The so modified first word is used for selecting the actual receiver thread by matching the word with the message criteria. When a matching receiver thread is found, the user supplied message, potentially including memory mappings and strings, is transfered from the sender thread to the receiver thread using an L4V2 IPC.

4.2.1 Receiver Preparation

Before explaining how a message is sent from one thread to another using the Generalized-Mappings IPC mechanism, I will show you how a receiver thread prepares itself for message reception. Figure 4.3 illustrates the receiver preparation.

At first the receiver thread prepares the message buffer wherein the message should be received. For a short IPC the "buffer" consists of the two direct words. Then the thread chooses the message criteria of messages it wants to receive by specifying the bit values and the number of significant bits. Now the library's GM-IPC receive function is invoked (1).

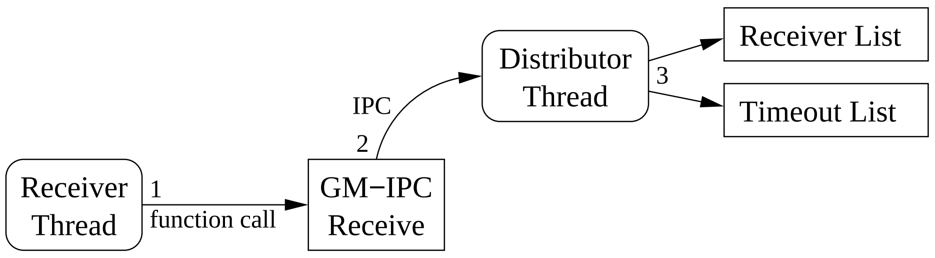

Figure 4.3: The receiver prepares for message reception.

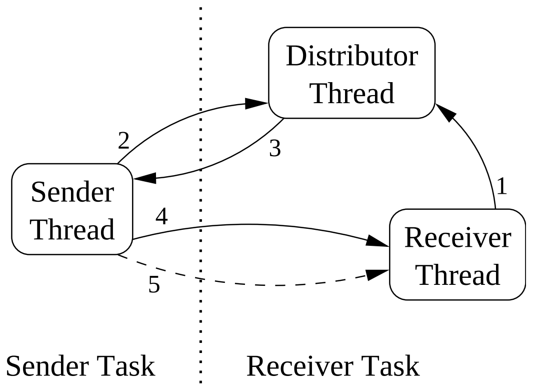

Figure 4.4: Control flow for a GM-IPC send operation.

The receive function builds up a receiver structure containing the message criteria and the timeout for the receive operation. Other fields of the structure are omitted here, they are introduced in the subsequent sections when these fields are actually used.

Then the receive function sends a pointer to the just created structure to its task's distributor (2). Here a pointer can be used, because the receiver and its distributor reside in the same task. The pointer allows the distributor to change the structure's fields directly without additional IPC overhead. This mechanism is used for the transfer of the initial message as described in the next section.

The distributor enqueues the receiver thread in its receive list (3) and waits for the next request[1].

After sending the message to its distributor, the receiver thread does an L4V2 open wait with the passed message buffer. The benefit of this open wait is that the distributor does not need to notify the receiver thread about the sender's L4V2 thread Id when a matching sender is found. The receiver is able to receive the actual message from any sender.

4.2.2 Send Operation in Detail

A thread of an application wants to send a message to another task using the Generalized-Mappings IPC mechanism. Therefore the thread prepares the message (in L4V2 format). Further on it sets up the capability descriptor with the receivers capability Id and the initial first message word. Now the thread invokes the GM-IPC send function provided by the developed library.

The control flow and the invoked L4V2-IPC operations are illustrated in Figure 4.4. Thereby, the first IPC (1) is the afore described receiver preparation.

The send function checks the validity of the send capability referred by the given capability Id. Appropriate fault handling is done if the send capability is invalid: Depending on the invoked function, either

[1]Actually, the distributor first checks the send list for a waiting sender whose message matches the message criteria. When a matching sender is found, the receiver is not enqueued. Instead the sender is dequeued and notified. See also Section 4.2.2.

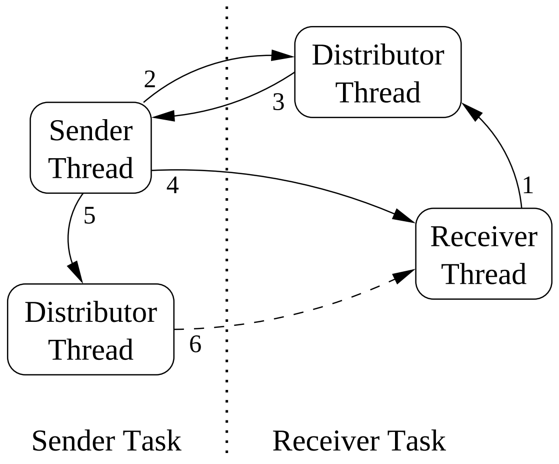

Figure 4.5: Control flow for a GM-IPC call operation.

the error code *invalid destination* is returned, or a capability-fault IPC to the capability pager is generated. From a valid capability the principal Id is taken and applied to the initial message word from the capability descriptor. Also the L4V2 thread Id of the GM-IPC distributor thread inside the receiver task is extracted from the capability.

Now the library sends an L4V2 IPC, consisting of the modified initial message word and the timeout for the send operation, to the receiver's IPC distributor (2).

The distributor uses the received initial word for selecting the actual receiver thread. Basically this is done by matching the initial word to the message criteria of every waiting thread. However, with optimized algorithms (see Section 4.4) it is possible to skip some unsuitable receivers. In case that no matching receiver thread was found, that is the initial word did not match any receiver's message criteria, the distributor enqueues the sender in its sender list and waits for the next request.

A matching receiver is dequeued from the receiver list. The initial word is passed to the receiver thread by copying the word into the receiver's message criteria. This works, because both the receiver thread and the distributor thread reside in the same task. Then the distributor replies the receiver's L4V2 thread Id back to the send function (3).

Finally the send function now sends an L4V2 IPC with the actual message to the receiver (4). This message consists of the two register words and potentially a memory buffer. These message parts are not modified by the GM library. The error code of this final IPC is delivered back to the invoker of the send function.

The purpose of the dashed dashed message (5) I will explain at the end of the next section.

4.2.3 Implementing an Atomic Call

L4 servers often reply to their clients using a send timeout of 0. This protects the server from denial-of-service attacks by malicious clients that send a request but do not wait for a reply. For the above reason the call IPC operation of an L4 kernel switches a client thread atomically from the send to the waiting state after message transfer, but before the server is invoked. The server can now handle the request and send the reply with timeout 0, because the client is already waiting for the message.

The same server behavior should be possible for servers and clients communicating with the GM-IPC mechanism. Therefore the library's GM call operation, more precisely a combined GM send and GM receive operation, must be implemented in a way that it also switches atomically from the send to the receive state. Otherwise the client may not be receiving when the server replies.

The major challenge during implementing such an atomic switch at user level was the interaction and synchronization between all involved threads: the sender, the receiver, and the sender's distributor. The distributor of the receiver is unproblematic, because it is used only during the sender's send stage for receiver selection. Figure 4.5 illustrates the interaction between the involved threads and the IPC messages sent between them. Again, (1) is the preparation of the receiver

The main constraint for an atomic call operation is that the receiver's receive operation cannot finish until the sender is in receive state. I implemented the following solution: After preparing the message to be sent, the capability descriptor and the message criteria, the sender invokes the library's send-and-wait function. The send part works exactly as described in Section 4.2.2 (IPCs (2), (3) and (4)). In principle, the receiver could now start servicing the request. However, this would violate the atomic-call constraint. Hence, the receiver's receive operation does not return immediately, instead it waits for an *IPC-complete* message. The send-and-wait function now prepares the receive state. The receiver structure is build up (see 4.2.1). An additional field therein is set to the receiver thread's L4V2 thread Id. The distributor will use this field for sending the IPC-complete message. Now the send-and-wait function calls the sender's distributor for receiver enqueuing (5). The distributor usually enqueues the now receiving sender. Afterwards the distributor sends the IPC-complete message to the receiver thread (6), the Id is taken form the receiver structure. Now the sender is in a proper receive state and the receiver can start processing the request.

Another implementation detail: The server can use both functions, receive and send-and-receive, when waiting for a message. Therefore both functions have to implement the afore described logic. However, when every receive function awaits an IPC-complete message, what happens when a basic send is invoked? Here, the sender does not call its distributor for receiving a message. Therefore the distributor cannot send the IPC-complete message to the receiver. My implemented solution is that the GM-send function itself sends the IPC-complete message to the receiver thread subsequent to the actual data message. This IPC-complete message is the dashed fifth message in Figure 4.4.

4.2.4 Implementing Timeouts for IPC

In L4, the IPC operation has timeouts for both the send and the receive part. The timeout specifies, how long a thread is willing to wait for the communication partner's invocation of the IPC operation. If the message transfer is not started until the timeout hits, the IPC operation is canceled and returns the error code *send timeout* to the sender, respectively *receive timeout* to the receiver.

Timeouts are used for several purposes: Firstly, a server protects itself from hostile clients by replying with a send timeout of 0. The reply will fail, if the client is not already in the receiving state. This prevents a DoS attack where the server is blocked by hostile clients that query the server but do not wait for the responses. Secondly, servers use timeouts to specify the maximum time they wait for an event. For example, a window server wants to awake at least once per second, even when no other requests are issued. Thirdly, timeouts are used to wait a certain amount of time. The sleep function is implemented using a receive with a timeout.

For these reasons, the implemented GM-IPC mechanism must also provide timeouts. The timeouts of the underlying L4V2 IPC cannot be used directly, because this could lead to improper results: For a send operation the send timeout would specify the time to wait for the receiver's distributor but not, as intended, the time until the actual receiver is ready. Also a consistent library state has to be kept. Only threads really waiting to send or receive should be in the library's lists.

Therefore I implemented timeouts for GM IPC inside the distributor thread. I will describe now, how timeouts for a GM-IPC send operation are implemented. Timeouts for the receive operation are implemented analogously.

The GM-IPC send functions already have a timeout parameter. This timeout is sent to the distributor thread when calling it for receiver selection. The distributor tries to find a matching partner as described. When no matching partner is found, the timeout is inspected. If the timeout is 0, the distributor sends a timeout error back to the sender. Then the sender aborts the send function returning *send timeout* to the caller. Otherwise the distributor enqueues the sender into the sender list. When a timeout other than ∞ was specified, the absolute[2] hit time is calculated and the sender is enqueued in the timeout list too. Thereby the timeout list is kept sorted by ascending absolute timeouts.

[2]In L4V2 timeouts are relative to the current time.

The distributor implements timeouts by waiting for new requests with a receive timeout of 10 ms. There is nothing special with 10 ms, it is just a tradeoff between accuracy and additional load due to timeout handling. However the used Fiasco kernel seems to handle timeouts also every 10 ms.

When the distributor receives a message, or when its receive timeout hits, the first timeout is checked. If it is less than the current time, timeout handling is done. Thereby the timed out thread is dequeued from the timeout list and from the sender list, respectively the receiver list. Then a timeout error is sent back to this thread. These steps are repeated until the first timeout is later than the current time or, of course, the timeout list is empty. After timeout handling the distributor processes the received request, if there was one.

4.2.5 Capability Mapping and Unmapping

The mapping of capabilities during GM IPC is done purely inside the library. In analogy to memory mappings the sender specifies a capability flexpage it wishes to send inside its message. The receiver specifies a receive window where it is willing to receive capabilities. After the receiver selection the capabilities from the sender's flexpage mapped to the receiver according to the rules from [2] if the flexpage and the receive window are of different size. Afterwards the L4V2 message transfer is started as described in Section 4.2.2. During this message transfer the sender's capability-flexpage descriptor is transfered as raw data by the underlying L4V2 kernel. The receiver can use the received capability-flexpage descriptor to determine which capabilities have been mapped into its capability space.

Unmapping of send capabilities, thus the revocation of communication channels, is a key functionality of Generalized Mappings. In today's L4, unmap is a system call. Since the library capsulates the handling and mapping of send capabilities, it has to provide also an appropriate unmap functionality for these capabilities. Like the GM-IPC functions, the library-provided unmap takes the same parameters as the standard unmap: the flexpage to unmap and an unmap mask. When the flexpage is a send-capability flexpage, the library handles the unmap internally, i. e. traversing the mapping database of each capability within the flexpage and unmap the capability from all tasks it was mapped to. As expected, unmapping of send capabilities works transitively as well. Other flexpage types, ordinary memory or IO flexpages, are forwarded to the L4 kernel's unmap system call. However, the *complete space* is special cased by the library's unmap: It first unmaps all send capabilities. Then the kernel's unmap is invoked, which unmaps all memory pages.

4.2.6 Capability Fault Handling

Another important functionality the library provides is the handling of capability faults. Capability faults occur when a capability Id referring to an invalid capability is used for an IPC operation. The two types[3] of fault handling are to send a capability-fault IPC to the pager or to abort the faulting IPC. For the latter case the library provides an extra send function which returns an error code. The standard send function does the first type of error handling.

To handle capability faults a task needs a capability pager. Because of the different handling of memory faults — they are detected by the L4 kernel and handled using L4V2 IPC — and send-capability faults, which are detected by the library and handled using GM IPC, the L4V2 pager of the faulting thread cannot be used as capability pager. Thus an extra pager for handling send-capability faults is needed. In the prototype the capability pager is per task. I decided so after analyzing the usage of the per thread memory pager. The memory pager was only changed when a region manager is used. The region manager then forwards the page fault to the appropriate pager. However, the same for capabilities can be achieved by supplying a wrapper around the error returning GM-IPC send functions.

The program can set the send-capability pager using a library provided function which takes a capability Id and the initial first message word as parameters. The mechanisms described in Section 3.2.7 are not feasible, because the prototype does only IPC management. For task and thread management the functionality of the underlying L4 kernel is used.

So when the GM send operation detects an invalid capability, it generates a capability fault message and sends that message to the task's capability pager. After the pager replied an appropriate capability

[3] For discussion refer to Section 4.2.6.

mapping, the failed send operation is restarted. If the pager capability is invalid as well, the faulting thread is shut down.

4.3 Matching Requirements

In Generalized Mappings IPC the receiver thread of a message sent to a task is selected based on the message content. A thread waiting for a message specifies a pattern, the message criteria, on its wait operation. An incoming message must match the specified criteria to be delivered to that thread. Refer to Section 3.2.6 for a detailed discussion.

The basic algorithm to find receiver thread for an incoming message is to compare the message with the message criteria from all waiting threads. When the comparison succeeds for a thread that thread is a valid candidate for message reception. Since the message criteria is a variable-length bit string, which is compared with the beginning of the message and the comparison ignores the tailing message bits, this type of comparison is called a *prefix match*.

The main challenge during receiver selection is to find the thread with the best matching message criteria from all valid candidates. That is to find the thread which specified the longest matching message criteria. So algorithms used in GM receiver selection must be able to search the longest matching message criteria (the longest matching prefix) for an incoming message.

A common example for longest prefix matching is the routing decision for packets in IP networks: For an incoming packet a router selects the next router, thus the outgoing interface, based on the packet's destination. From its routing table the router searches the entry with the longest matching prefix to the desired destination.

In contrast to prefix matching, many search algorithms perform *exact matching*. Here two keys are compared and the comparison only succeeds when both keys are equal and of the same length. For example consider the password for a computer login: To gain access, the typed password must be exactly the same — char-by-char — as the stored password. When you type more or less characters, the login will be denied.

The problem with exact matching for GM receiver selection is that in many cases the specified message criteria will shorter than the incoming message. So exact matching will not find a valid receiver candidate. Cutting the message to the criteria length can prevent the failed comparison. However, cutting once to a fixed length does not solve the original problem because threads inside a task can specify different criteria lengths.

4.4 Receiver Selection Algorithms

This section describes possible algorithms for receiver selection. The benefits and the problems when using the algorithms for prefix matching are discussed. Concrete numbers for the implemented algorithms are presented in Chapter 5.

For search time complexity the following notation is used: n is the number of waiting threads. w is the maximum length of the message criteria the threads specified for their wait operation.

4.4.1 Linear List Search

The first implemented algorithm for receiver selection is a linear search through all waiting threads. I chose the linear search, because I did not expect any implementation difficulties and I wanted to run my test applications. So I got a working GM IPC path without sophisticated receiver selection.

Linear search works directly on the distributor's client-thread descriptors of my library. The descriptors of all waiting threads are chained into a linear list. On an incoming message the whole list is traversed to find a suitable receiver. The receiver thread with the longest matching prefix is chosen as the message recipient and dequeued from the list. When multiple receivers specified the same prefix and length (e. g., they implement a thread pool) the first one found is used.

In principle, linear search can be used with any other matching algorithm, because the matching algorithm is applied to the message specification of every receiving thread. However, in this thesis only prefix matching is used.

The time needed for finding the actual receiver thread for an incoming message depends on the number of waiting threads. So the time consumed for receiver selection is $O(n)$. Inserting a new receiver thread into the list is done in $O(1)$, because new receivers are always enqueued at the beginning of the list. The enqueuing does not have an influence on the search operation, because all receiver-thread descriptors are searched through thereby.

4.4.2 Binary Trees and Tries

Binary Tree

A binary tree is a data structure that can be used for storing data keys and finding them later. The tree consists of nodes, each node stores a data key and two pointers to the left and the right subtree. A search for a key starts at the root node of the tree. The search key is compared with the node's data key. When the keys are equal, the search was successful. Otherwise, depending on whether the search key is greater or less than the node key, the search continues on the right or on the left subtree. Searching the binary tree terminates either on a successful comparison — the key was found — or when branching to a non-existing subtree. In the latter case the searched key was not found in the tree.

The shape of a binary tree depends on the insertion order of the keys. Therefore binary trees can become unbalanced. In the worst case the tree degenerates to a linear list. This is especially a problem, because the worst case can occur frequently in practice, for example when the keys are inserted in sorted order.

The search complexity of a perfect binary is $O(\log n)$. However, unbalanced trees need more time to search a key, up to the worst case of $O(n)$.

Another problem of a binary tree is that both the insert and the search operation use an exact match between the search (or insert) key and the node key. In contrast, the receiver selection for GM IPC has to find the thread with the longest matching message criteria. Padding a short message criteria with zero bits (or any other known bit combination) does not work, because the respective bits of the incoming message are not known a priory. The padding bits likely have a wrong value, so the expanded criteria will not match the incoming message.

Binary Trie

Both problems, the unpredictable shape and the applicability of prefix matching, can be addressed with a *binary trie*. Let me first describe the basic concept of a binary trie[4]. Thereafter I will discuss prefix matching and further optimizations.

Like the binary tree a binary trie consists of nodes with a data key and two subtree pointers. The branching decision, however, is not determined by comparing the keys themselves. Instead the individual bits of the search key are used. A search starts with the search key's most significant bit at the root node. The left branch is taken on a `0` bit, the right branch on a `1` bit. Trie traversal continues with the next bit. The basic search scheme is considered successful, when the bits of the search key run out and the data key of the reached node is valid. Otherwise the searched key was not found in the trie.

The basic search scheme does exact matching, as a binary tree too. Fortunately, the trie can be adapted for prefix matching: When the finally reached node from the basic search scheme does not contain a valid data key, the search backtracks, potentially up to the trie root. Only when no valid node is found on the walked path, the search terminates unsuccessful. Otherwise the backtracking ensures that the data key with the longest matching prefix is found. A more efficient implementation omits backtracking by remembering the last visited valid node during tree traversal. The remembered node can be returned directly when the finally reached node has no valid data key.

The shape of a binary trie depends on the inserted keys but not on their insertion order, because for the branching decision the individual key bits are used and not complete key value. So the search time for a key in a binary trie does not depend on the number of keys nor their insertion order. Only the key length

[4]The term *trie* is derived from retrieval [6].

is important: The search complexity of a binary trie is $O(w)$. This also holds for searching the longest matching prefix.

Path Compression

Unlike binary trees, binary tries do not become unbalanced. However, the tries are often still too deep: The depth of a leaf node is equal to the length of its key in bits.

Long chains of nodes with only one child node occur frequently in tries. Such a chain without branches indicates that all keys following the chain have an identical prefix. The idea of path compression is to convert these chains into a single node. Thereby the skipped branches, thus the skipped key bits from the original chain, are recorded in that node. The resulting path compressed trie is less deep than the corresponding binary trie except in the rare occurring worst case.

Path compressed tries are also called *prefix tries*. An example for a path compressed trie is the PATRICIA trie [23] used in text searching algorithms and the 4.3BSD routing table [26].

With prefix free keys a PATRICIA trie with n stored keys has n leaf nodes with the keys, called external nodes, and $n-1$ internal nodes. Each internal node has exactly two child nodes. The average depth of a PATRICIA trie with m nodes is $\log m$, while the average depth of the binary trie is the average key length.

When the keys are not prefix free, the external nodes are not necessarily leaf nodes. They can also occur within the trie. In contrast to internal nodes, external nodes within the trie can have only one child node. The number of internal nodes in such a trie is at most $n-1$. The average trie depth remains $\log m$.

The rarely occurring worst case for a path compressed trie is when the keys have increasing length and shorter keys are prefixes of longer keys. Then the resulting trie is a sorted linear list.

Implementation

In my library I did not implement pure binary tries for matching the incoming message with the receiver's message criteria because of the expected chains without branches. Nodes in such chains require additional storage. They also increase the search time for a receiver because of the needed bit tests per node and also due to the increased cache working set: The more nodes have to be tested the more cache is needed and therefore more application cache lines are destroyed.

A prefix trie does not suffer on these two problems, therefore I implemented it. I did not use different node layouts for internal and external trie nodes, because the given message criteria of one thread can be a prefix on another thread's message criteria. Then an external node exists inside the trie and not only at the leafs. Another reason is that I do not have to test for different node types during trie traversal when searching or inserting a key. Also, on insertion of a new waiting thread whose specified message criteria is the same as the key of the reached internal node, there is no need to convert the internal node to an external node.

Thus, a trie node consists at least of two pointers to the left and to right subtrie, a pointer to a thread list with all threads waiting for the reached prefix and the bit number to test for the next branching decision. The problems that arise now are how to represent the skipped bits and how to test them. One solution would be to store the skipped bits inside the trie node as a variable-length bit string and compare them before each branch. However, variable-length bit strings are often inconvenient to handle. Trie implementations used in virtual memory management for translating virtual addresses into physical addresses delay the comparison of the skipped bits until a leaf node is reached [27]. The entire key is stored in the leaf node or in a table outside the trie, for example in a frame table. After reaching a leaf node the entire key is compared with the stored key at once. When the keys are not equal, some of the skipped bits were different and no valid address translation exists.

Unfortunately, the described algorithm cannot be used for GM, because of its assumption: The algorithm assumes a prefix free trie which is true for memory. This assumption does not hold for the message criteria of waiting threads in GM IPC, where one thread can specify a longer criteria starting with exactly the same bits that are another thread's shorter criteria. In such a case for an incoming message the best matching criteria has to be found for IPC handshake and message delivery.

In my prototype a message criteria has at most 32 bits. So I decided to include two additional fields into my trie nodes: the entire node key and a mask for the valid key bits. The mask is precomputed from

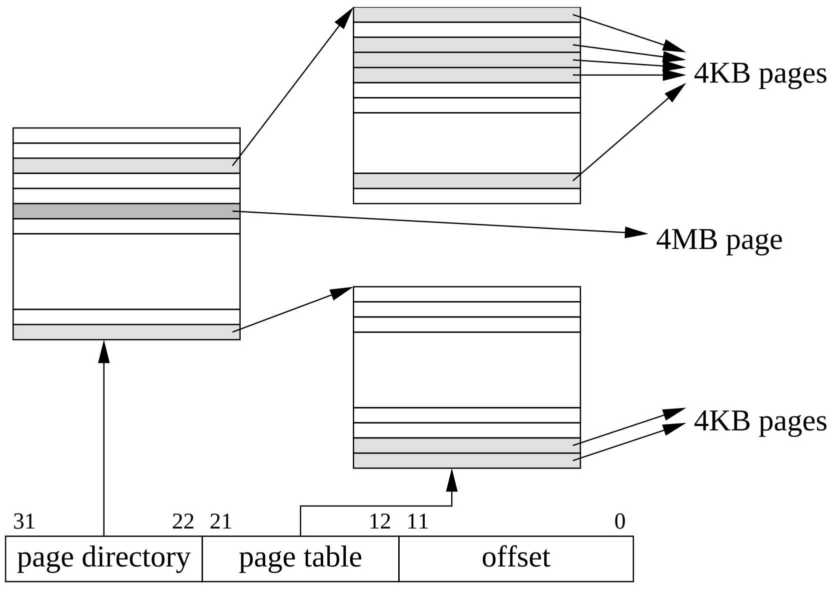

Figure 4.6: Page table structure of the Pentium MMU.

the branching bit number during node insertion. Therefore, a trie node needs 24 bytes memory on the x86 architecture. Together with additional padding my trie nodes use 32 bytes memory, which is exactly an L1 cache line on the Pentium processor.

During trie traversal the node's mask is applied to the search key. Then the masked search key is compared with the node key. When both keys are equal trie traversal continues on the left or on the right subtrie depending on the search key's bit value on the branching bit position. Otherwise, if the keys are unequal, the trie traversal is stopped and the search algorithm backtracks to find the last matching node.

To achieve higher performance I avoided backtracking by slightly extending the trie traversal algorithm: When the key comparison succeeds and the node's thread list is valid (i. e., not a `NULL` -pointer) my algorithm memorizes the node as the best matching node so far. Later, on a node where the search key does not match the node key, the algorithm straightly goes back to the memorized node, which is automatically the best matching node.

4.4.3 Page Table like Structures

During discussions about possible structures for fast receiver selection often the idea came up that structures similar to page tables could be used for searching a matching receiver. I will discuss now the difficulties when using page tables as structures for receiver selections.

In principle, page tables are generalized tries with a greater base than 2: In each level of the table there are existing more than two child-table pointers. For branching to the right child table more bits than one bit are used. For example, the two-level page table used on the x86 architecture [8] is a trie with the base 1024. The first level is called *page directory*, the second level *page table*. From the given virtual address the first 10 bits are used as index into the page directory to get the corresponding child page table. The next 10 bits are an index into the page table to get the physical page. An extension introduced with the Pentium processor are super pages. For a super page the page directory entry point not to a page table, but to a 4MB super page. Hence, the Pentium page tables are a trie with variable key length. Because a page directory entry points either to a super page or to a page table, the address space is always prefix free. Figure 4.6 illustrates the address translation for 4KB pages and 4MB super pages.

The usage of multi-level page tables for translating virtual into physical addresses has several benefits: In an uncompressed binary trie only one bit is used per level, so many single bit tests are necessary to find the corresponding physical page (e.g, 20 tests are needed on the x86). Instead, in page tables many bits are used to find the next table level (10 bits on x86). So the number of memory accesses and the actual translation time is reduced.

Like a trie, multilevel page tables allow holes in an address space: When no pages are allocated inside a large region (on x86 a 4MB region aligned to 4MB) no page table is needed for that region, the entire region is marked invalid in the page directory. This mechanism helps to save memory for page tables by allocation only the actually needed page tables. So smaller tables offer the opportunity of saving table memory because they can support smaller holes. However, smaller tables per level means more table levels, increasing the translation time. Therefore the table size is a tradeoff between the memory consumption for the page table and the translation time.

For larger address spaces, e. g. the virtual address space on 64-bit architectures, more page-table levels or larger tables per level are needed. Assuming a page size of 4K, a table entry needs 8 Byte (52 bits page address and 12 bits for permission, attributes, etc.), so a 4K frame holds 512 Entries. A multi-level page table will have 6 levels for address translation. Such a deep table is inefficient for most tasks, because only small portions of the address space are actually used. So tables on intermediate levels often contain only one valid entry.

Guarded Page Tables (GPT) [16] skip these intermediate levels by using a guard representing the skipped address bits. Effectively, skipping bits during translation is applying path compression to the page table. Hence, GPTs are generalized PATRICIA tries with a radix greater than 2. Like in tries, path compression helps to save memory and translation time. Another benefit from GPTs is the support of different page sizes with $size = 2^n$. Thereby, like hierachical page tables, guarded page tables support prefix free keys. Furthermore, with GPTs the size of a page table in a table level can vary depending on the page distribution in the corresponding memory region. Larger tables are used for densely populated regions, smaller ones for sparsely populated ones. In both cases the skipped bits help to reduce the table depth.

L4 implementations for the MIPS [3] and the Alpha processor use GPTs for address translation.

Page Tables or GPT for Receiver Selection?

Page tables and guarded page tables used for address translation are designed for prefix free search keys. Prefix free means that inside a memory region covered by a super page (e. g., a 4MB region aligned to 4MB on x86) no translation for a 4K page can exist. This property is enforced by hardware (x86) or by software (L4 on MIPS [3, 27]). Both implementations use a single bit test to distinguish whether the translation reached a valid super page or a next level page table.

However, the message criteria used for selecting a receiver thread in the GM-IPC mechanism has not the property of prefix free keys. Rather, the longest key from all keys matching the incoming message is searched. So a table entry can contain both a valid key with a link to a waiting thread and a link to a next level table for possible longer keys. Hence, unmodified standard page tables cannot be used because they only support one link type per entry at a time. Another problem I described already with tries is also present on page-table like structures: Since the key space is not prefix free, the search algorithm has to backtrack when no matching key is found on deeper table levels. To save search time backtracking can be optimized as in my implemented trie (Section 4.4.2).

The main problem I see when using page table like structures for receiver selection is the arbitrary length of the message criteria, especially in combination with the prefix keys. With standard page tables it is difficult to support arbitrary key length because a fixed number of bits is used as an index into the table. However, a waiting thread of a GM task can specify a fewer number of bits as its message criteria. The obvious solution of using less index bits, therefore using smaller tables, leads to a binary trie.

One solution to support any possible length is a table layout as shown in Figure 4.7. Here a table level consists of several tables: The first table provides the linkage to the next table level. The first k bits from the key are used as index into this table (in the example k is 4). For each possible length l smaller than k a table for threads specifying l significant bits as message criteria exists. In the example there are tables for threads with three, two, and one significant bits containing eight, four, and two entries. You should note the difference between entries of different tables marked with the same number: The first entry of the 3-bit table means `000` , i. e. three zero bits, while the first entry of the one-bit table stands for `0` , one zero bit. The final one-entry “table” is a bit special, it is used for threads waiting for all messages regarding the reached table. For the top-level table this field is used for threads invoking an open wait. In deeper levels

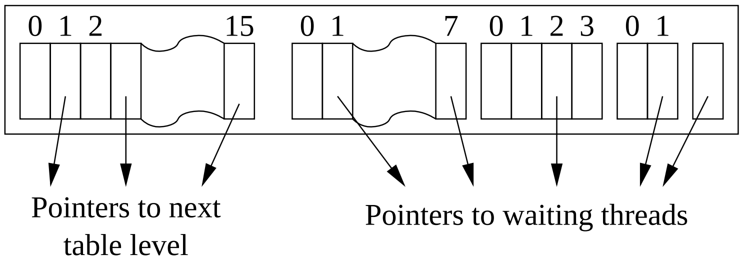

Figure 4.7: A possible page table structure for receiver selection.

threads are enqueued in this one-entry "table" when the length of their message criteria is a multiple of k: In every table level above k bits are used as index into the linkage part, so no significant bits from the criteria are left in the current level.

The problem with such a table structure is to find an efficient matching algorithm. When using a depth-first search, that is to walk down to the next table level until a leaf is reached, backtracking has to be used when no matching thread is found in the leaf-table's subtables. Backtracking can be expensive depending on the table size and depth. Using a breadth-first search means that on every visited table level all k l-bit subtables are searched for matching threads before the algorithm steps to next table level. The problem for fast searching I see here is the linear search in every table level. However, as in my trie, backtracking can be avoided by remembering a matching entry form a higher table level. Another possible optimization is to stop the linear search when a subtable contains a matching thread: In all later subtables can only exist threads with shorter matches but the thread with the longest matching criteria is searched for.

Because of the discussed difficulties and mainly due to the lack of time I did not implement table like structures for receiver selection.

4.4.4 Hashing

During my work I also thought about hashes and hashing algorithms for fast receiver selection. The main idea of using hashes for receiver selection is the hopefully constant low search time. If it is so, hashing would allow selecting the actual receiver thread in time $O(1)$ regardless of n or w. However, the total overhead will be slightly higher than in current L4 because the calculation of the hash function takes some short time.

The problem I see with hashing is the exact match semantics. For keys with different lengths the hash function will certainly generate different hash values. So in general the complete first word of an incoming message will likely not generate the same hash function output as an a few bits short message criteria that matches the message otherwise. Also in GM-IPC not only one arbitrary matching thread is searched, the thread with the longest matching message criteria has to be found.

A naive idea to solve the problem of different criteria lengths is the expansion of shorter a message criteria to a common length with a fixed known pattern. The problem hereby is that the extra bits will likely not have the same value as the corresponding bits of the incoming message. Therefore the hash function will compute two different outputs for the expanded message criteria and the incoming message, thus indicating no match between the two. So no valid receiver thread is found even if there is one, but with a shorter message criteria.

Another, more feasible, idea is the usage of one hash per existing message-criteria length. Before applying the hash function, the incoming message masked according to the criteria length of the individual hash. But how to find the hash with the best matching criteria for a particular message? Here a second-stage algorithm for selecting the right hash is needed. However, the search time for a key then depends not only on the hash (hopefully $O(1)$), but also on the second-stage algorithm. So the time consumption of that algorithm is also important.

One feasible algorithm is a linear search through all existing hashes. As there is one hash per criteria length, up to w hashes are searched degrading the time for receiver selection from the expected $O(1)$ to

$O(w)$. When searching the hashes in ascending criteria length order, always all hashes must be searched through because the last hash with the most specific message criteria might contain a matching receiver thread. Like in my trie, a hash containing a match is remembered until a hash with a longer criteria has a matching thread inside. So it is guaranteed that the thread with the longest matching message criteria is selected for message reception.

In contrast, searching the hashes in descending length order can stop on the first hash containing a match. All unsearched hashes can contain only less specific matches with a shorter message criteria. So descending search is faster than ascending search except when only the shortest length hash contains a match or no match is found at all.

So in the worst case, searching through all hashes, hashing is slower than a trie (also $O(w)$) because the hash calculation is more complex than the trie's bit test for branching. For the average case an estimation is more difficult. Hashes may be faster, depending on the actual implementation, when only few different criteria lengths each with a lot of threads with different criteria exist. Then a trie will have many branches toward the leafs consuming many bit tests while only a few hash calculations are necessary. Descending search will also be faster than a trie when the wide majority of incoming messages is for waiting threads that specify a long message criteria. Here the first searched hash will contain a valid receiver thread — no further hashes are searched.

However, for the described difficulties with hashes, the unclear search time, and due to the limited time for this thesis I did not implement hashes for receiver selection.

Chapter 5

Evaluation

In this chapter the implemented Generalized Mappings IPC mechanism will be evaluated quantitatively. However, detailed end-to-end IPC benchmarks, for instance using the standard ping-pong benchmark, are rather useless because the prototype implementation is a user-land wrapper library: The library layer introduces additional latency in terms of the total consumed time and also in terms of cache usage. Therefore I created a set of microbenchmarks to measure the important parts from the GM-IPC path. With the microbenchmarks I can avoid the library's additional IPC overhead.

Evaluation Environment

All microbenchmarks were done on an Intel Pentium processor (P54C, CPUID 5:2:B) with 133 Mhz on an Intel i430 FX chipset based mainboard (Asus P54TP4). The processor has an 8 KB 2-way associative data cache with a cache-line size of 32 bytes. The data TLB has 64 entries for 4 KB pages and is 4-way associative. These TLBs are flushed on an address space switch. Further on, the board is equipped with 64 MB RAM and 256 KB unified 2nd level cache. The network card, an Intel Ethernet Pro 100, is only used for booting the machine.

The microbenchmarks ran on the Fiasco microkernel extended by a system call for flushing the caches from user level. Besides the benchmarks the standard components from the L4Env [12], the name server, the log server and the dataspace manager, were running. All used software was compiled with gcc 2.95.4.

All times presented in the next sections are in processor cycles. They where measured using the processor's internal timestamp counter, which is read using the `rdtsc` instruction [9] — excluding the overhead of that instruction.

Experiments

I created two microbenchmarks that cover the two fundamental parts of the Generalized-Mapping IPC path: the capability usage for allowing and authenticating IPC, and the receiver selection for finding the actual receiver thread. The first part, the capability access, I account to the sender side. The second part, receiver selection, clearly depends on the concrete receiver thread structure, so it is accounted to the receiver side. Both parts together build the IPC handshake.

The first microbenchmark measured the additional costs when using a capability for authorizing an IPC. As my library is a prototypic design the capability access and check is written in C and not highly optimized. Therefore a measurement of that path would result in a relatively high numbers indicating slow capability checking and thus much overhead. So I decided to implement an optimized path for benchmarking the capability testing overhead in assembly language like it will be used in an in-kernel implementation. The measurements of that path and a discussion follows in Section 5.1.

With the second microbenchmark the performance of the implemented algorithms for receiver selection is quantified. Here I created a multi-threaded server task and a single-threaded client. The server threads are waiting for messages. Thereby each server thread uses a unique message criteria. The different thread's message criteria have a length of 24 bits and are distributed uniformly among that 24-bit address space.

The client is in a loop. Within each iteration it calls a randomly chosen server thread. Inside my library I instrumented the receiver selection to measure the time it takes until the receiver thread is definitely found. The consumed time is averaged over 10000 client calls. For the implemented receiver selection algorithms the numbers are presented in Section 5.2 followed by a discussion.

5.1 Performance on the Sender Side

When a task sends an IPC message a kernel with the new Generalized-Mappings IPC mechanism has to execute the following steps:

1. calculate the kernel representation from the sender specified send capability to access the actual capability
2. check the validity of the send capability
3. apply the principal Id from the capability to the message
4. figure out the destination task to start the receiver selection in that task.

The send capabilities are stored in a large array indexed by the capability Id. A naive implementation would use `0...CAP_MAX` as capability Id. To access the actual capability, the capability Id is multiplied with the kernel's capability size and the base of the array is added. However, multiply operations are quite slow compared to other arithmetic operations on most processors. Also a bounds check with `CAP_MAX` has to be done to avoid using arbitrary memory regions as send capabilities.

Liedtke showed in [15] that a large array in the virtual address space is feasible for fast addressing of thread control blocks in L4 when using a clever encoding scheme for thread Id's. The same mechanism can be used for addressing the send capabilities: To calculate the address of the capability's kernel representation the user supplied capability Id is masked and the base offset of the send-capability array is added. Consequently only two basic arithmetic operations are necessary: an `and` for the mask application and an `add` with the base offset. Both the mask and the base offset are constant.

The next step, the capability validity check, consists of two operations, one test operation and a conditional branch. The test operation checks the valid field of the capability structure. When the test fails, the branch is taken and capability error handling takes place. In the common case, the capability is valid, the branch is not taken. Since modern microprocessors assume conditional forward branches as not taken and fetch the instructions immediately following the branch into the pipeline, the common case is not slowed down significantly.

In the third step the principal Id is applied to the message. Thereby bits used by the PID are masked out from the message. Then the PID is added to the message. Both fields, the mask and the PID come from the internal capability representation. The mask is calculated during the map operation and stored in the capability for performance reasons during capability usage. So PID application are also two operations.

Now let me consider two cases: The fast case occurs when all accessed data resides in the cache and the TLB contains a valid address translation for the used send capability. In the slower second case the caches and the TLBs are empty.

In the optimal fast case all described operations can be executed at maximum speed. Since a valid address translation exists in the TLB an all data (the whole capability structure) resides in the cache no additional cycles besides the normal instruction execution are needed. In the microbenchmark I measured 6 cycles for the optimum case.

In the slower case the first two operations, the calculation of the actual capability structure, runs at the same speed as the fast case because no memory is referenced. However, for the validity check the capability, and thus the memory, has to be accessed. No valid address translation for the capability structure can be found in the TLB so the hardware[1] starts to walk through the page table to find a valid translation for updating the TLB. As the memory address used by the capability cannot be found in the cache, additional cycles are needed to access the real memory. My send capabilities are using 32 bytes memory, which is

[1]The x86 architecture has a built-in hardware page table walker.

exactly one cache line on the used processor. So neither additional cache nor additional TLB misses occur during the next operations for PID application. I measured 64 cycles for this case with my microbenchmark.

Cache	TLB cold	TLB hot
cold	64	40
hot	31	6

Table 5.1: Time needed on sender side in cycles.

Table 5.1 summarizes the time used for capability access and principal Id application with different cache and TLB states. One TLB entry and one data cache line are used during the operations. However, the cache line is only read, no writes to the capability are issued.

Besides the aforementioned considerations on data caches and D-TLBs also the influence on the instruction cache and I-TLB must be considered. For the I-TLB I do not expect additional TLB misses because the IPC path is performance critical. So the IPC path will be page aligned in a kernel designed for maximum performance. One I-TLB miss can occur when the IPC is invoked. All further code accesses are covered by the then existing page translation. The instructions for capability testing and PID application, illustrated within this section, use 20 bytes of memory, roughly two thirds of a cache line. Therefore I expect the usage of up to one additional cache line in the instruction cache due to the capability use.

5.2 Performance on the Receiver Side

On the receiver side I account the costs for selecting the actual receiver thread from all waiting threads within the receiver task. In the optimal case the taken time is constantly low guaranteeing low overhead and high IPC performance. However, as discussed in Section 4.4, none of the implemented algorithms allow receiver selection in constant time. The consumed time depends either on the number of waiting threads or on the used message criteria and its length.

Another important criteria besides the consumed time is the algorithm's influence on the cache and the TLB. The more memory accesses at different memory locations are performed by the receiver selection algorithm, the more cache lines are used. Thus, more application data is purged from the cache that usually must be reloaded after the IPC. However, cache reloading from the main memory is an expensive operation on modern processors because memory accesses are magnitudes slower than cache accesses. The same argumentation holds for reloading the TLB. So a small cache working set is important for the overall performance as Liedtke points out in [19].

Table 5.2 shows the time consumed for receiver selection with the implemented algorithms. Each algorithm was measured with hot and with cold caches. The impact on the TLB is not considered, because on the used hardware the TLB is flushed on each address space switch resulting in an empty TLB when receiver selection starts.

Linear List

As described in Section 4.4.1 my list implementation works directly on the distributors client-thread descriptors, which are chained into a doubly linked list. During receiver selection every element from the list touched for matching the incoming message against the stored message criteria and also to find the next list element.

The time consumed for finding the receiver thread with the best matching message criteria depends on the number of list elements, thus the number of waiting threads. The time grows with the number of waiting threads because every single thread's criteria is matched. Therefore the linear search algorithm does not scale for many threads: Receiver selection is becoming slower and slower the more threads are waiting for messages.

Another Problem is the number of memory accesses and thus the pressure on the cache. For every list element three memory accesses too the thread descriptor are done: Reading the message criteria, reading the mask for the significant bits and reading the pointer to the next list element. Due to the thread-descriptor

Threads	Trie cached	Trie uncached	List cached	List uncached
1	113	156	106	126
2	148	189	133	158
3	162	222	140	188
4	171	243	156	219
8	198	270	215	319
10	219	286	228	369
16	254	308	391	514
32	329	359	804	929
50	369	419	1122	1380
64	390	449	1754	1875
75	403	459	2209	2167
100	466	475	3130	3055

Table 5.2: The time for receiver selection in processor cycles.

design and usage — one thread descriptor layout is used for both send requests and receive requests — the memory accesses are scattered across two cache lines. Therefore the cache usage also grows linearly with the number of threads. For 100 server threads nearly the complete first-level cache is used (6400 bytes out of 8192 bytes).

For the described reasons it is not surprising that linear list search consumes a lot of cycles when many threads are waiting for messages. However, selecting a receiver with searching a linear list was implemented only as a quick hack to get the library working without expecting high performance. Nevertheless the algorithm can be useful for tasks using only few threads, namely single threaded tasks or multi-threaded tasks with up to approximately four threads.

Trie Search

Another algorithm I implemented for receiver selection is the binary trie described in detail in Section 4.4.2. In contrast to linear search the trie consists of nodes, which are stored separately from the thread descriptors. A thread descriptor is only accessed when trie search was successful and therefore a matching thread was found.

The time to search a key inside the trie depends on the number of trie nodes that have to be accessed until the search key is found. Therefore the trie shape, especially the maximum and the average depth, is important. The trie shape and the depth are depending on the used keys and their distribution across the key space.

For benchmarking an uniform distribution of the keys, the message criteria, was chosen. The resulting trie depth for n threads is $\lceil \log n \rceil$. The trie consists $n-1$ internal nodes and n external nodes with waiting receiver threads as leafs.

For finding a key (i. e., matching the incoming message), $\log n$ nodes from the trie are accessed. Consequently, since the computation on a node is constant, the consumed search time is also proportional to $\log n$. As expected, for many threads the trie algorithm is much faster as the linear list.

In terms of cache usage the trie also outperforms the list. A trie node uses 32 bytes so it fits into one cache line. Since $\log n$ nodes are accessed during trie traversal, $\log n$ cache lines are used. However, when a leaf node is reached the thread descriptor from the waiting thread is accessed, which always costs two additional cache lines.

As you can see in table 5.2, for up to four threads the trie is slower than the list. The reason for this is the higher overhead on each trie node: In the list the next-element pointer is accessed directly while in the trie a bit test followed by a conditional branch is executed to decide whether the left or the right subtrie is searched next.

Chapter 6

Conclusion

In this thesis I developed a capability-based communication control and corresponding IPC mechanism for the L4 microkernel. The requirements for the IPC mechanism are presented in Section 3.1. A prototypical implementation proved the feasibility both from a performance and functional point of view.

The Generalized-Mappings communication mechanism uses explicit communication channels between tasks. These channels are authorized with task-send capabilities introduced in Section 3.2. Possession of such a capability allows its owner to send messages to the task the capability refers to. A task without send capabilities to a particular task cannot send messages to that task. Therefore, send capabilities are an effective way to restrict the ability of a task to communicate to other tasks.

In Section 5.1 I showed that the overhead introduced into the time-critical IPC path causes by the additional indirection when using capabilities is fairly low. In the optimal case this indirection and the needed calculation costs only 6 additional cycles. In comparison with the total IPC costs of 215 cycles these additionally incurred costs are negligible. This case, however, assumes that all needed data resides in the cache and a valid address translation exist in the TLB. In the worst case one additional cache miss and one additional TLB miss occur in the IPC path when accessing the capability, for discussion see Section 5.1.

I also discussed how I extended the mechanisms L4 provides for memory management at user level, the mapping and unmapping of pages, to apply them to send capabilities. Thereby the map operation allows for the delegation of rights: A task can map its send capabilities to another task. Afterwards both tasks can send messages to the receiver task specified in the capability. In contrast to other capability based systems, such as Amoeba or EROS, our approach provides an effective way for selective revocation of capabilities: the unmap operation. Unmap enables a task to revoke capabilities it previously mapped to other tasks. Afterwards these tasks cannot use the unmapped capability for sending messages anymore.

The developed IPC mechanism provides local names for both senders and receivers. Using local names enable several advantages over globally unique names: They allow late binding from names to actual communication partners. The name of the communication partner remains the same when the partner is exchanged, for example when a server task is restarted due to an update.

The flexibility of this new communication mechanism I showed in Section 3.3. First, I showed how an intercepting task, for example a debugger or a filter, is inserted into an existing communication channel. Thereby the insertion of the interceptor can be transparent to the involved tasks. Second, I emulated the current L4 approach with globally unique thread Id's on top of local names and capability authenticated communication to show backwards compatibility.

With Generalized Mappings IPC the receiver thread of a incoming message is chosen by the kernel based on the message content. On a receive operation, a thread specifies a pattern, the message criteria, an incoming message must match to be delivered to that thread. The kernel matches the message to the specified message criteria and delivers the message to the waiting thread if the criteria matches the message. This mechanism, however, slows down the IPC performance because in the timeframe of this thesis no matching algorithm could be found that has a constant low selection time regardless of the number of receiver threads. The consumed time of all algorithms discussed in Section 4.4 depends either on the concrete number of threads in the receiver task, or on the length of the message criteria. It remains an open question, whether there exist matching algorithms for selecting the receiver thread in constant time, or whether another mechanism for receiver selection has to be found.

6.1 Future Work

In-Kernel Implementation

The Generalized Mappings IPC mechanism should be implemented inside a L4 kernel in future. An in-kernel implementation does not suffer from the additional library overhead. Also security requirements regarding the communication restrictions can be met in an in-kernel implementation.

Appendix A

Appendix

A.1 Glossary

Task Is a collection of resources. Often a task is used for accounting purposes. A task is also a protection domain.

Thread An activity inside a task, an abstraction of a CPU. A thread can use the resources its task owns.

Memory The virtual address space of a task.

IPC Communication between threads.

Message A message is sent from one thread to another with the IPC Operation. A message contains a direct part, indirect strings (optional) and flexmappings (optional). The direct part and the strings are simply copied from the sender to the receiver. Flexmappings are inserted into the appropriate tables (e. g., memory mappings into the page tables, send capabilities into the send-capability table).

Task-Send Capability The right to send a message to a task. Send capabilities are objects provided and protected by the kernel. User applications use them for invoking IPC and can map them to other tasks. The send capability refers to the task actually receiving message. The receiver cannot be altered by the task possessing the capability. However, more bits can be appended to the kernel protected PID of the capability during the map operation.

Open Wait A thread waits for any incoming message.

Group Wait In contrast to open wait, a thread waits for a given message criteria. Incoming messages to the task that match the given criteria are delivered to this thread.

Closed Wait As in group wait, a thread specifies a message criteria the incoming message must match. However, in contrast to group wait, the specified message criteria can only be sent by one sender task[1].

Principal Id The kernel authenticated part of a message. The PID is stored inside the kernel as part of a task-send capability. It is not accessible by the capability-possessing task.

Message Criteria The receiver specifies a message criteria for messages it is willing to receive. An incoming message that matches the criteria is delivered to the waiting receiver.

Flexmapping A generalized flexpage. Resources (e. g., memory pages, or task-send capabilities) are distributed between tasks with messages containing one or more flexmappings.

Resource Fault a generalized page fault. The kernel sets up a fault IPC to the pager on behalf of the faulting thread.

[1]provided that the senders pager does not try to impersonate

Pager A resource-fault message is sent to the pager of the faulting thread. The pager resolves the fault and maps the requested resource.

$\rightarrow B$ A send capability that can be used to send messages to task B.

$A \rightarrow B$ Task A owns a send capability to task B.

A.2 Acronyms

ABI application binary interface

ACL access control list

API application programmers interface

D-TLB translation look-aside buffer for data accesses

DoS denial of service

GM generalized mappings

GM IPC generalized mappings IPC

I-TLB translation look-aside buffer for instruction fetches

IPC inter process communication

PID principal Id

TCB trusted computing base

TLB translation lookaside buffer

VM virtual memory

Bibliography

[1] M. J. Accetta, R. V Baron, W. Bolosky, D. B. Golub, R. F. Rashid, A. Tevanian, and M. W. Young. Mach: A new kernel foundation for unix development. In *USENIX Summer Conference*, pages 93–113, Atlanta, GA, June 1986.

[2] A. Au and G. Heiser. L4 user manual. UNSW-CSE-TR 9801, University of New South Wales, School of Computer Science, Sydney, Australia, March 1999.

[3] K. Elphinstone. Virtual memory in a 64-bit micro-kernel, 1999.

[4] The Fiasco microkernel. URL: `http://os.inf.tu-dresden.de/fiasco/`.

[5] Fiasco-UX, the Fiasco User-Mode Port. URL: `http://os.inf.tu-dresden.de/fiasco/ux/`.

[6] Edward Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960.

[7] Michael Hohmuth. The Fiasco kernel: System architecture. Technical Report TUD-FI02-06-Juli-2002, TU Dresden, 2002.

[8] Intel Corp. *Pentium Processor User's Manual, Volume 3: Architecture and Programming Manual*, 1993.

[9] Intel Corp. *Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference*, 1999.

[10] Trent Jaeger, Kevin Elphinstone, Jochen Liedtke, Vsevolod Panteleenko, and Yoonho Park. Flexible access control using IPC redirection. In *7th Workshop on Hot Topics in Operating Systems (HotOS)*, Rio Rico, Arizona, March 1999.

[11] Trent Jaeger, Jonathon E. Tidswell, Alain Gefflaut, Yoonho Park, Kevin J. Elphinstone, and Jochen Liedtke. Synchronous IPC over transparent monitors. In *Proceedings of the 9th workshop on ACM SIGOPS European workshop*, pages 189–194, Kolding, Denmark, 2000. ACM.

[12] L4 Environment Concept Paper. Available from URL: `http://os.inf.tu-dresden.de/l4env/doc/l4env-concept/l4env.ps`, June 2003.

[13] Butler W. Lampson. Protection. In *Proceedings of the 5th Princeton Conference on Information Sciences and Systems*, pages 437–443, Princeton, March 1971. Reprinted in *Operating Systems Review*, 8(1):18–24, January 1974.

[14] J. Liedtke. Clans & chiefs. In *12. GI/ITG-Fachtagung Architektur von Rechensystemen*, pages 294–305, Kiel, March 1992. Springer.

[15] J. Liedtke. Improving IPC by kernel design. In *Proceedings of the 14th ACM Symposium on Operating System Principles (SOSP)*, pages 175–188, Asheville, NC, December 1993.

[16] J. Liedtke. Page table structures for fine-grain virtual memory. *IEEE Technical Committee on Computer Architecture Newsletter*, pages xx–xx, xx 1994. also published as Arbeitspapier der GMD No. 872, German National Research Center for Computer Science (GMD), Sankt Augustin, 1993.

[17] 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.

[18] 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.

[19] J. Liedtke. μ-kernels must and can be small. In *5th International Workshop on Object Orientation in Operating Systems (IWOOOS)*, pages 152–155, Seattle, WA, October 1996.

[20] J. Liedtke. Toward real μ-kernels. *Communications of the ACM*, 39(9):70–77, September 1996.

[21] Jochen Liedtke. L4 version X in a nutshell. Unpublished manuscript, August 1999.

[22] Linux website. URL: `http://www.linux.org`.

[23] Donald R. Morrison. PATRICIA—practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968.

[24] R. M. Needham and R. D.H. Walker. The cambridge CAP computer and its protection system. In *Proceedings of the 6th ACM symposium on Operating systems principles (SOSP)*, pages 1–10. ACM Press, 1977.

[25] J. S. Shapiro. *EROS: A Capability System*. PhD thesis, University of Pennsylvania, April 1999.

[26] Keith Sklower. A tree-based packet routing table for berkeley UNIX. In *USENIX Winter Conference*, pages 93–104, Dallas, TX, January 1991.

[27] Cristan Szmajda. A new virtual memory implementation for L4/MIPS.

[28] A. S. Tanenbaum, S. J. Mullender, and R. van Renesse. Using sparse capabilities in a distributed operating system. In *6th International Conference on Distributed Computing Systems*, pages 558–563, Cambridge, MA, May 1986.

[29] Andrew S. Tanenbaum, Robbert van Renesse, Hans van Staveren, Gregory J. Sharp, and Sape J. Mullender. Experiences with the amoeba distributed operating system. *Communications of the ACM*, 33(12):46–63, 1990.

[30] Version X.0 specification of the L4 API. URL: `http://www.l4ka.org/documentation/files/l4-86-x0.pdf`.

[31] Microsoft windows website. URL: `http://www.microsoft.com/windows`.

[32] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. Hydra: The kernel of a multiprocessing operating system. *Communications of the ACM*, 17(6):337–345, July 1974.