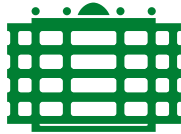


Fakultät für Informatik

Juniorprofessur Echtzeitsysteme

Diplomarbeit

Multiprocessor Support for the Fiasco Microkernel

Autor: Sven Schneider

Betreuender Hochschullehrer: Dr. Robert Baumgartl

Betreuer: Dipl.-Inf. Michael Peter

Abgabedatum: 01.08.2006

Sven Schneider

Multiprocessor Support for the Fiasco Microkernel

Diplomarbeit, Technische Universität Chemnitz, 2006

Contents

I. Introduction **9**

II. Fundamentals and Related Work **11**

1. L4 Mikrokernel **11**
1.1. Basic L4 Concepts . . . 11
1.1.1. Address Spaces . . . 11
1.1.2. Threads . . . 12
1.1.3. IPC . . . 12

2. Fiasco **13**

3. Design Proposal for MP Extensions for L4v2 API **13**

4. General Scalable L4 Multiprocessor Implementation **13**

5. Earlier Fiasco MP Port **14**

III. Analysis **17**

6. Design Goals and Aspects of Analysis **17**

7. User Model **17**
7.1. Design Space . . . 17
7.1.1. Threads . . . 18
7.1.2. Memory . . . 18
7.1.3. Communication . . . 19
7.1.4. Interrupts . . . 22
7.1.5. New Cross Processor Operations . . . 22
7.2. General Global Model . . . 23
7.3. Split Model . . . 23

8. Multiprocessor User Environment **24**
8.1. L4env . . . 24
8.2. L^{4}Linux . . . 25

IV. Design **29**

9. General Global Model vs. Split Model **29**

10. Partitioned User Model **30**
10.1. Memory and Tasks . . . 30
10.2. Threads . . . 30
10.3. Communication . . . 31
10.4. Interrupts . . . 31
10.5. Cross Processor Operation . . . 31

11. Kernel Design **31**
11.1. Data Ownership and Data Access . . . 32
11.1.1. Processor-Local Global and Static Variables . . . 32
11.1.2. Allocated Pages . . . 33
11.1.3. Thread Control Blocks . . . 34
11.2. Synchronization . . . 34
11.2.1. Traditional Single Processor Synchronization . . . 34
11.2.2. Spinlock . . . 36
11.2.3. Message Box System . . . 36
11.3. Cross Processor IPC Path . . . 37
11.3.1. Requirements . . . 37
11.3.2. Design Aspects . . . 38
11.3.3. Cross Processor IPC States . . . 38
11.3.4. Atomic Transition from Send to Receive Stage . . . 40
11.4. Miscellaneous Cross Processor Functionality . . . 40
11.4.1. Kernel Console . . . 40
11.4.2. Low Level Memory Allocator . . . 41

V. Implementation **43**

12. Kernel **43**
12.1. Message Box System . . . 43
12.2. TCB-Area . . . 44
12.3. Remote IPC-Path . . . 45
12.3.1. Message Box Requests . . . 46
12.3.2. Atomic Sequences . . . 46
12.4. Debugger Synchronization . . . 48

13. Basic Services **50**
13.1. σ_0 . . . 50
13.2. Roottask . . . 51

14. Test and Debugging **51**
14.1. Test Framework . . . 51
14.2. Test Methods . . . 52
14.2.1. Modulated Delay Loop . . . 52
14.2.2. Time Window . . . 52
14.3. Tests . . . 53
14.3.1. Low Level Tests . . . 53
14.3.2. IPC-Path Tests . . . 54
14.3.3. Interference Tests . . . 55
14.3.4. Performance Tests . . . 55
14.4. Debugging Illegal TCB Accesses . . . 55
14.5. Found Bugs . . . 55

VI. Evaluation **57**

15. Low Level Functionality **57**
15.1. Cache Performance . . . 57
15.2. Cross Processor Synchronization . . . 58

16. IPC Performance **59**
16.1. Cross Processor IPC 59
16.2. Influence of Workload on Local IPC 60
16.3. Local IPC on MP-Kernel vs. IPC on UP-Kernel 61

17. Interrupt Latency **61**

VII. Conclusion and Future Work **63**

VIII. Summary **65**

IX. Appendix **67**

A. Glossary **67**

B. Bibliography **68**

List of Tables

1. Cycles Needed For Cache Protocol State Transitions 58
2. Cycles Needed For Cross Processor Synchronization 58
3. Expected Round Trip Times for Cross Processor IPC 59
4. Measured Round Trip Times for Cross Processor IPC 60
5. Influence of Modification of Processor Local IPC Path 61

List of Figures

1. Cross Processor Communication Using Proxies 20
2. Scenario of Synchronous Communication Using Proxies 21
3. Setup for a L4env Multiprocessor Application 24
4. L^4Linux Multiprocessor Setup 26
5. Partial Duplication of the Uniprocessor Kernel 32
6. System Global Variables 32
7. Processor Local Variables by Array Indexing 33
8. Processor Local Variables by Memory Mapping 33
9. Remote Procedure Call of the Message Box System 36
10. State Diagram of Cross Processor Send Operation 39
11. Message Box Matrix and Single Message Box 43
12. Mapping Structure with TCB Master Directory 45
13. States of a Debugger Instance 49
14. Scenario for Delay Loops to Detect Race Conditions 52
15. Scenario for Time Windows to Detect Race Conditions 53
16. Influence of Different Workload on Local IPC (Cross Sibling) 60
17. Influence of Different Workload on Local IPC (Cross Package) 61
18. Interrupt Latencies at Different Workloads 62

Part I.

Introduction

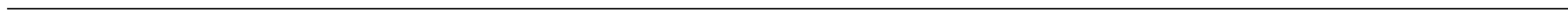
In the last years the parallelism in computer systems increased so far that today one finds multiple processors even in off-the-shelf computer systems, i.e. standard PCs. But there are kernels that were designed solely for uniprocessor platforms and have special properties, just like the Fiasco microkernel [10] with its realtime properties. These kernels could evade the support of multiprocessor systems for a long term. However, they are more and more confronted with the new multiprocessor systems.

But kernels for multiprocessor systems are more complex than those for uniprocessor systems. This is because a new category of intricate problems needs to be addressed: shared mutual data need to be synchronized. Naïve approaches such as global spinlocks result in not acceptable performance degradations. On uniprocessor systems this was not a problem because atomic sequences could be easily implemented. For kernels that have realtime properties it gets more complicated since complex in-kernel protocols might be needed that could also impair the performance.

Uniprocessor kernels use assumptions that are not true in multiprocessor systems. These assuptions lead to a simpler kernel because kernel objects can combine differnt semantics. However, some these semantics should be distinguished in multiprocessor systems for performance reasons. Porting an existing uniprocessor kernel to multiprocessor systems means a high effort because the combined semantics must be split again. So wide parts of the basic kernel infrastructure might have to be redesigned and the higher level parts of the kernel have to be adapted to use the new infrastructure correctly. An example for such kernel objects is the thread lock of Fiasco that combines IPC and thread synchronization.

The goal of this work is to port the Fiasco microkernel to multiprocessor systems in a way that it keeps its realtime properties and can be taken as basis for a multiprocessor port of L^4Linux [8, 19, 20]. The approach following in this work is *partial duplication* of the uniprocessor kernel in order to reduce the development costs. The most of the existing kernel infrastructure need not to be touched, only a small enhancement for explicit cross processor operations is needed.

The following part (pp. 11) will present fundamentals and related work. Afterwards the design space for the user model is elaborated in the analysis part (pp. 17). In the design part (pp. 29) the user model that is to be implemented is chosen and the changes in the kernel design are presented. After that, some implementation details are described in the implementation part (pp. 43). In the evaluation part (pp. 57) the performance of the implementation is measured and rated. In the following part (pp. 63) conclusions are drawn and future work is outlined. Finally, the document closes with a short summary (pp. 65).



Part II.

Fundamentals and Related Work

1. L4 Mikrokernel

The design philosophy of microkernels is to implement basic mechanisms within the kernel and to move the policies to the user land. For example, paging-strategies are implemented in user level. This means the actual handling of page faults in user address spaces and page substitution algorithms are implemented at user level. The kernel only provides a mechanism to map pages with certain access rights. Another example are security policies. The decission which application is allowed to use which resources, is made in the user land. When granting or revoking an access right the kernel is informed about the change. The kernel is responsible to enforce the decission.

Microkernels are used for security relevant applications because they provide a very small (and therefore probably easier verifyable) trusted kernel which is able to separate trusted (security relevant) applications from untrusted applications. Further the minimality of micro kernels is helpful for realtime systems as there is a smaller part of the system which is to made realtime capable. These advantages can be combined with the usage of a commodity operating system, such as Linux, on top of the microkernel alongside trusted and realtime applications.

The first L4 microkernel was developed by Jochen Liedtke [21, 22]. It was implemented for the Intel IA32 architecture. Meanwhile there is a variety of differents API specifications available for L4 (e.g. V2 [23], X0 [24], X2 [6] (a.k.a V4), L4.sec [5, 18]). There are also various implementations with different goals: E.g. performance (L4Ka::Hazelnut [2]), portability (L4Ka::Pistachio [3]), security (L4.sec), embedded systems (Pistachio-embedded [4]), or kernel-resource management [7].

1.1. Basic L4 Concepts

This section describes L4 concepts that are important for this work in a simplified way. It especially focuses on the L4v2-API.

1.1.1. Address Spaces

The address space abstraction represents protection domains. It means address spaces are the means to isolate user level applications from each other. Access rights to resources are given to address spaces. The resources can have an address space local identifier under which it is accessed, e.g. access rights for physical memory frames: The access rights for pages can be writing allowed, read only, or no access possible. The local identifier of the frames are the virtual addresses where these frames are mapped. Other resources and their access rights that are managed by this concept can be communication end-points (e.g. L4.sec) or IO-ports (e.g. on the IA32 architecture).

In the V2-API, address spaces also hold a fixed set of threads; thus the terms address space and task are used as synonyms in this context.

There is a root address space (σ_0) which contains all physical memory. This address space is the starting point for mapping memory from one addres space to another. To map and unmap pages between address spaces three different operations are provided.

The **grant** operation transfers pages to other address spaces. The address space that grants a page has no access to this page after the granting (because the page is removed from the granting address space).

The **map** operation makes a page visible in another address space. The main difference between map and grant is that the page is still accessible in the address space that maps the page.

The **flush** operation removes a page that was mapped beforehand from all destination address spaces. This includes all indirect mappings (and grantings) that have their origin in the flushed page.

These three operations are initiated by threads belonging to the mapping, granting, or flushing address space. For the map and grant operations the destination address space must accept the pages. Therefore, the map and grant operations are part of the IPC system call. This means the pages are sent from the source thread to a destination thread The flushing of a page need not to be confirmed by any address space that holds the page.

Further every address space has a chief. Only the chief of a task is allowed to create it (i.e. starting the first thread), to delete it (i.e. killing the first thread), or to handover the address space to another chief. Strictly speaking, clans and chiefs are part of an own concept. This concept was introduced for access restriction but was later dropped because it was to inflexible. The only remnants of this concept is the control right over tasks described above.

1.1.2. Threads

In L4 the thread abstraction plays following roles: It is the entity to which time slices are designed in the scheduling model. It represents the execution context of an activities. And finally, in L4v2, it also represents an end-point of communication.

The scheduling in L4 is based on static priorities. Therefore each thread has a priority assigned. This assignment is done from user land by the system call `schedule`. Threads of a higher priority preempt threads of lower ones. Therefore only threads of the same (highest) priority are running. These threads are scheduled round robin.

Since threads also represent the execution state it is possible to retrieve or to change it. The system call that is used for this in `lthread_ex_regs`.

In some APIs (e.g. V2 in contrast to L4.sec), threads are addressed directly as the target of communication operations. There are also virtual threads, which only have this role. These threads represent external interrupts.

1.1.3. IPC

IPC denotes the only means for threads to communicate and synchronize across address space boundaries. The IPC operation is synchronous: Both sender and receiver must reach the synchronization point before the actual message transfer begins. This procedure is called rendezvous, the synchronization point is also called rendezvous point.

Messages that are transfered by the IPC operation can contain machine words, memory mappings, a buffer of machine words, a number string buffers, or any combination.

Page faults are mapped to IPC: When a page fault in an user address space occurs an implicit IPC operation is initiated. The IPC message contains a short information about the pagefault and is sent to a previously registered thread (a.k.a. pager). The pager resolves the pagefault. It can send memory mappings in the reply message. Only after the reply message was sent the execution of the faulting thread is continued.

For the communication with the virtual interrupt threads a special protocol was designed that acts on top of the IPC communication mechanism, Threads can attach to an external interrupt and subsequently can receive IPC messages that signal that an interrupt occurred.

2. Fiasco

The Fiasco microkernel [10] has been developed at TU Dresden, the kernel implements the APIs L4v2 and L4x0. The support of the L4v4/x2 is currently in development.

The design had a strong focus on realtime properties [9]. Fiasco ensures interrupt latencies, i.e. the time from the occurence of the interrupt to the actual activation of the thread that received the interrupt message if that thread is the one with the highest priority at this time. It also bounds the time needed for an IPC operation transfering a message containing up to two machine words if the other thread is ready to receive the message. Further, the L4 scheduling model was extended to provide Quality Assuring Scheduling [27].

3. Design Proposal for MP Extensions for L4v2 API

Völp [29] proposes a multiple processor model and kernel design for the V2-API. This model has a strong focus on the compatibility to the existing single processor model so existing applications can execute without any functional restrictions. Other goals were the flexibility, generality, performance, transparency for non-enlightened applications, and non-transparency for the enlightened applications.

However, the transparency does not extend to the implicit properties of the L4 scheduling model. The former assumpion was that only threads with the same priority are running because threads that have a lower priority are preempted and there are no threads with a higher priority (they would preempt threads with the regared priority). This assumption is broken because the kernel scheduler works only processor local and thus priorities only have local relevance.

The non-transparency for multiprocessor aware applications is achieved by extending the current uniprocessor API. So a multiprocessor aware user level scheduler can now place the threads to selected processors and applications can find out whether an IPC operation was cross processor by inspecting a formerly unused flag in the IPC return value.

For the kernel internal synchronization a mailbox system is used. Incoming messages are signalled by an inter processor interrupt. To keep the interrupt handler as short as possible it only sets up an in-kernel thread that will handle the message itself.

The requirements of the user model are chosen in a way that will not induce the slightest inconvenience for user level applications even if it means much more complexity for the kernel. Further this approach targets an unknown general workload and not a specific workload like L^4Linux.

4. General Scalable L4 Multiprocessor Implementation

Uhlig describes in [28] a complex synchronization scheme that ensures the scalability of the microkernel. His goal was to ensure high performance and scalability on a microkernel at the same time, i.e. for applications that do not use all processors the synchronization costs should be not higher as in a system where the unused processors were not present. For this he proposes dynamic adaption of the synchronization strategy and granularity as solution.

He describes a mechanism of runtime adaption of the synchronization method which is called *adaptive locks*. For every locking granularity a special lock exists. There are two locking scopes. One is processor local, the other is cross processor. The application hints the kernel which processors shall have direct access to the lock. If this is only a single processor the processor local scheme is used, if there are more the cross processor scheme must be used. All processors that have no direct access to the lock must prepare a message and send it to a processor that is allowed to access the lock and therefore can act as a proxy.

The mapping database was modified so only portions of a mapping tree could be locked. Such a portion is called *lock domain*. The advantage is that now there can be independent accesses to different parts of the mapping tree, which naturally increases the scalability. *Adaptive locks* are used to for synchronization within a *lock domain*.

Because the principle to keep the policy out of the kernel should be followed, it is necessary that the applications give the kernel a hint which locking scheme is to be used. This requires the applications to be enlightened and to know its own communication patterns. The applications that fulfill this requirement are called well designed and are explicitely favored. The well designed applications use as much local communication as possible in order to avoid the expensive cross processor operations. This also means that servers provide a server thread on each processor and applications only communicate with the local server thread. So after startup, the cross processor communication is mainly used for synchronization.

Uhligs work aims at a general model with general purpose workload and has intention that all existing applications will work, though with a possible performance loss. Adaptive locks, which consitute a complex synchronization mechanism, were implemented in order to exploit the difference of various synchronization methods and to use the one that fits best for a particular situation. Further global datastructures like the mapping database are made more complex due to the introduction of *lock domains*. Realtime properties are not subject of his work.

5. Earlier Fiasco MP Port

The basic approach that was followed in an earlier multiprocessor port for Fiasco [25] was to extend the locking infrastructure. It was deemed it would be a small change because Fiasco was already fully preemptible.

In uniprocessor systems the thread lock ensured that the locked thread will not be scheduled. Since only a single thread could run at a the same time it could be concluded that the locked thread is not running and will not become running. But in multiprocessor systems every thread that is *ready* could be *running* at another processor. For this it must be detected whether the locked thread is still running. The detection of the current remote execution requires expensive synchronization operations. This also effects processor local operations. Further, a new thread state was introduced in order to distinguish between a thread that is *ready* to run and one that is actually *running*. Because of the new state deadlocks become and issue: If two threads try to lock each other at the same time both would have to wait for the other the become not running. So for this situation it must be possible to become not running while waiting for another thread to do the same. But to really avoid the deadlock this must be done by a single thread otherwise the thread locks would never be released.

Further, in Fiasco, thread locks are helping locks, which implement wait-free synchronization using a helping scheme. Though it was possible to implement a cross processor helping scheme with realtime properties, it turned out that the substantial run-time overhead for the processor local case was prohibitive.

In this approach a general multiprocessor solution was targeted. The Fiasco uniprocessor kernel infrastructure had some implicit assumptions that did not longer hold for the multiprocessor environment. So the thread lock assumed that its effect were prompt. This behavior was given for uniprocessor systems where all operations were serialized.

It turned out that multiple aspects of locking were combined in one lock (the thread lock). On multiprocessor systems these aspectes are better addressed separately: The first aspect is IPC synchronization. This synchronization is to synchronize multiple sender threads that want to send a message to a single receiver thread and does not need the receiver to interrupt its execution.

The other aspect is the thread synchronization. This is needed for migration as the migrated thread must not be running. Another mechanism that needs this kind of synchronization is `lthread_ex_regs` as the current execution context only becomes visible and changeable when the modified thread is interrupted.

Part III.

Analysis

In the analysis part the trade off between functional requirements, preserving the realtime properties, avoiding runtime overhead, and implementation complexity is discussed. After the goals have been defined the design space for the user model is explored. Finally, a design for user land applications on top of the favored user model is proposed.

6. Design Goals and Aspects of Analysis

The main goal is to provide a user model that is well suited for a multiprocessor port of L^{4}Linux. Besides that the multiprocessor-supporting Fiasco microkernel shall still have the realtime properties of bounded interrupt latency, bounded execution time of the register value IPC operation, and its realtime scheduling model. However, it is not necessary that the model is transparent to multiprocessor unaware applications. It is accepted that the operation of such applications is impaired.

Therefore the aspects of analysis are how much does the multiprocessor modifications influence the performance of processor local operations, what complexity is added to the kernel, how much are the realtime properies influenced, and how much transparency is provided for multiprocessor unaware applications.

Because processor-local operations are expected to be much more frequent than cross processor ones, they should incur the smallest possible performance degradation. The realtime properties of Fiasco should also hold in the multiprocessor version of the kernel. For the fulfillment of this requirement complex realtime protocols might be needed which contradicts the wish for a low kernel complexity. If there are possibilities that fit equally to the intended workload of L^{4}Linux and make little difference in the implementation effort, the one that restricts multiprocessor unaware applications least shall be chosen.

7. User Model

In this section, the user model is described in more detail. After the discussion of the design space, the group of the commonly used multiprocessor user models for L4 is described. Finally an alternative user model is proposed.

7.1. Design Space

In the analysis of the design space, the unified and the split system view, their differences, and their implications for applications and kernel will be discussed. First, the ideas behind the two different system views are explained. Afterwards, these views are discussed for the different abstractions and mechanisms that are provided by the user model.

The *unified system view* provides full transparency; no abstraction that can represent a system processor is visible. On the one hand, this is very tempting because applications would need no modifications. But on the other hand, any kind of information about and control over the multiple-processor system is missing. User land has no influence on which processor a thread is running (i.e. thread placement). And even multiprocessor aware applications would have difficulties to avoid expensive cross processor operation and

this might not exploit the full parallelism provided by the system. So the strict usage of this view might result in an infeasible system. Therefore mechanisms to control the multiprocessor system, such as thread placement, and to retrieve location information should be introduced. This also introduces an inherent abstraction for system processors.

The *split system view* does not try to hide the system structure but exposes many of the low level details. The system partition is introduced as a new abstraction. Each system partition represents one system processor. So the system provides a split view to user land. The multiprocessor view is not transparent to multiprocessor unaware applications. Therefore the operation of those applications is restricted. If the model does not provide any mechanism that operate cross processor, it will be inconvenient even for multiprocessor aware applications.

7.1.1. Threads

In this section only the roles as schedulable unit and execution state representation are discussed. Threads acting as communication endpoint designator are discussed in section 7.1.3.

Scheduling

Using the uniprocessor L4 scheduling model, as descibed in 1.1.2, would result in global scheduling. As described by Völp [29], this is infeasible due to the runtime costs induced by the synchronization of the global runqueue and idling processors. Originally only threads with the same (highest) priority execute. If there are fewer such threads with this same priority than processors are available some processors will idle although there might be other threads ready for execution. So a processor local scheduling model is strongly preferred.

Execution State

A cross processor execution state manipulation via `lthread_ex_regs` means that the manipulated thread must be temporarily halted and thus the execution flow of another processor must be interrupted. This results in to higher kernel complexity as an infrastructure must be provided for this. The processor local operation might only incur a slight performance impact as it is possible to multiplex the cross processor and the processor local operation to the same interface.

7.1.2. Memory

Main options in the memory model are whether the address space abstractions can span across processor boundaries and whether to support cross processor mapping relations.

If the address space abstraction spans across processor boundaries some consistency management is needed to make sure that changes of memory mappings are visible on all processors. This influences cross processor mapping and flush operations as well as processor local ones because the address space could be used on remote processors, too. Even if an address space is only used on the local processor the mapping or flushing operation must first check whether the address space is used on another processor.

If the kernel does not support cross processor address spaces it is still possible to emulate them at user level by building multiple address spaces with a congruent mapping. This leads to a bigger resource foot print because a pagetable is needed for each task an application consists of. Furthermore, an overhead is

introduced because the memory must be mapped for each address space separately. But there are also the advantages that the kernel will be simpler and if cross processor address spaces are not needed and thus need not to be implemented the trusted computing base becomes smaller.

If address spaces are only a processor local abstraction, another question that arises is whether address spaces can migrate from one processor to another.

The other main option was whether to support cross processor mapping relations. If such relations shall be tracked by the mapping database, it must be modified to work system globally. This could be achieved either by using a simple locking scheme or by redesigning the mapping database for the multiprocessor system. The simple locking scheme results in additional overhead for processor local operations. And there also is the possibility of lock contention that is not well solved by this approach. A redesign of the mapping database that ensures that the performance loss of local operations is not too high might lead to complex synchronization protocols.

If no cross processor mapping relations shall be managed, there has to be a mapping database per processor that holds the processor local mapping hierarchy. There is an option whether to support the transfer of pages from one processor to another. If they are supported, an hierarchy origin, i.e. a σ_0-task, is needed on every processor. Due to the local-only mapping relations the question whether it is possible to pass a page from one processor to another, arises. For this the concerned pages must be removed from their current mapping hierarchy and inserted to the hierarchy on the destination processor. In this new hierarchy a new father must be found. Only σ_0 can be the father because it is always present and has always mapped all pages. But the mapping hierarchy will be flattened when transfering pages across processor boundaries and thus the hierarchical semantics of the mapping operation will be lost. Such a change of the *map* semantics is infeasible. Therefore the transfer of pages between processors must not be allowed. This transfer can happen if a task migrates from one processor to another or if a cross processor mapping operation is applied. Thus neither task migration nor cross processor *map* operations are supported.

7.1.3. Communication

A first variant that provides full transperancy for multiprocessor unaware applications is to support the same functionality for the cross processor communication as for the processor local one. However, this depends on the chosen memory model because in L4 the memory mapping operations are embedded in the IPC mechanism. Thus the complexity of a fully transparent IPC operation includes the complexity of transparent memory as descibed in section 7.1.2.

Another possibility is to limit the existing IPC mechanism to the local processor and to introduce a completely new cross processor mechanism. If multiprocessor unaware applications shall be able to communicate across processor boundaries an adaption layer is needed that hides the pecularities of the actual transport mechanism. This layer could consist of proxy threads in order to avoid the modification of multiprocessor unaware (legacy) applications. This will be refered to as proxy communication or proxy model.

Figure 1 shows a scenario with proxies used as adaption layer for the multiprocessor unaware applications. For the communication with other processor local threads, including the local proxy thread, the original IPC mechanism is used, which now is working only processor locally. The proxies use the new cross processor communication mechanism to forward the message to the right proxy that finally delivers the message using the local mechanism again.

Proxies might provide only an efficiently implementable subset of the communication interface of normal threads. Location transparency is only provided for the subset that was implemented. The variant that uses proxies leads to kernel enhancements for the cross processor operation as well as enhancements in unser level in the form of the proxy adaption layer. So not only the kernel but the whole scenario gets more complicated. This possibility will be referenced as proxy communication.

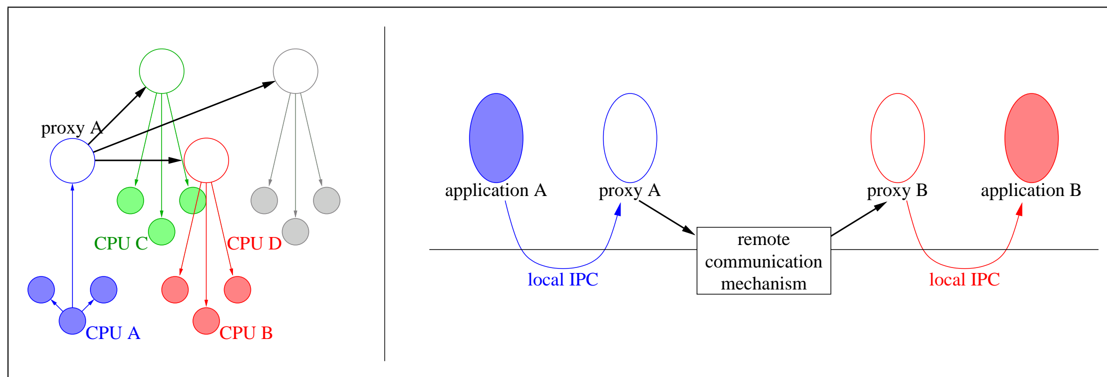


Figure 1: Cross Processor Communication Using Proxies

Finally, a third possibility lies between the both described before. The cross processor mechanism provides a very restricted functionality that is a subset of the one provided by the processor-local mechanism.

Both mechanisms are using the same kernel interface. If multiprocessor unaware applications only use this specific subset the cross processor mechanism is location transparent without any proxy at user level. Only the kernel complexity is increased by a dedicated cross processor mechanism and the multiplexing of two different mechanisms behind a common interface. A slight performance penalty is expected for the processor local operations due to the multiplexing. This and the first possibility (full transparency) will be references as flat communication.

Addressing Remote Threads

As in L4v2 threads also represent communication targets. They are addressed by the communication operation.

For the cross processor operation a system global id space for communication endpoint designators must be provided. If proxies are used this space only needs to be visible to the proxy threads and multiprocessor aware applications, the unaware applications might be ignorant of it. However, there is an addressing variant where the address space consists of system global thread ids and applications therefore are aware of it. In this variant the application could simply address the final recipient. If the communication target is located on another processor the kernel redirects the message to the local proxy thread that will forward the message to the right processor. This approach is location transparent, the application need no special knowledge about the location of its communication partner. The kernel must provide a redirecting mechanism which also slightly influences the processor local operations because the check whether to redirect the message must be made for every operation.

Another possibility is that the application threads send the message directly to the proxy. Here, a special proxy protocol that also includes the address of the final recipient might be needed. This address could be a global thread id or even a service indicator. The kernel need to know nothing about the actual addressing scheme that is used in the proxy protocol. This approach is very intrusive concerning the applications because the proxy protocol is not transparent. Further it might be the same effort to change multiprocessor unaware applications to use the special protocol compared to adapt them to use the new cross processor mechanism directly.

To achieve the transparency even if the message is addressed to the proxy it is possible to have for every remote thread that can be addressed a proxy thread that represents it. So by sending a message to a certain proxy thread the final recipient is implicitly addressed. This also means not all remote threads can be reached but only those for which a proxy thread exists as representative. This addressing scheme is partly

location transparent because an application needs not to know where a thread is running but at different locations a single thread might be known under different names. This degenerates to a model where the cross processor communication uses proxies threads as communication end points instead of application threads.

If no proxies are used and location transparency for multiprocessor unaware applications is needed the kernel must provide system global IPC destination addresses so they can be used for the direct addressing. In the case of L4v2 these are system global thread ids.

Synchronous vs. Asynchronous Communication

The first possibility is to provide a synchronous message transfer. This means there is a rendezvous between sender and receiver. Only if both participants are at the rendezvous point the message transfer is started. Parameters, namely timeouts, control how long to wait for the other at the rendezvous point. The return with a success status code means that the message was completely transfered to the receiver. As this is the L4 semantics this would be fully transparent for multiprocessor unaware applications.

However, if proxies are used, as described earlier in this section, kernel extensions are needed to support this semantics. *Figure 2* depicts a scenario how a synchronous message transfer could be established if

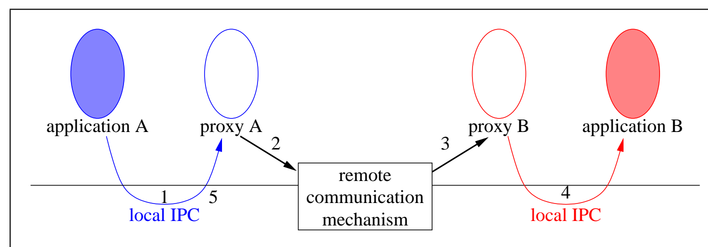


Figure 2: Scenario of Synchronous Communication Using Proxies

using proxies. An application thread at CPU A sends the message to its proxy (step 1). The send operation must not yet return because the message is not yet delivered to the final recipient. In case of proxies at sender and receiver side, the proxy uses the cross processor communication to deliver the message to the receiver proxy (steps 2, 3) which will deliver the message to the final recipient (step 4). When the message was accepted the acknowledgement must traverse the communcation chain backwards and finally unblock the local IPC operation of the sender (step 5).

For this scenario it is necessary to get the message from the sender without acknowledging it. The original IPC automatically acknowledges the message when it has been delivered. For this purpose the local IPC operation must be extended so a receiver can retrieve the message without acknowledging it. Another operation is also needed for the explicit message acknowledgement. This requires an additional thread state *wait-for-acknowledge*. This breaks the formerly atomic operation into two suboperations. Now, the `lthread_ex_regs` operation becomes very complex if applied to a participant of a cross processor communication because the ongoing operations of all participants have to be cancelled. A potential race between the actual delivery of the message and a cancellation must be serialized at the receiver. If it happens that the cancellation was not early enough it must not have any effect to the result of the communication. However, if no proxies are used, the synchronization is completely done in the kernel. Thus, the message can be transfered to the other processor without the need for retrieving the message by an extra IPC operation.

If the asynchronous semantics for the message transfer is used the cross processor operation will not wait until the message was delivered. The unblocking of the sender gets a weaker semantics meaning the proxy

only accepted the message for delivery and it is not yet clear whether the message will be successfully delivered compared to the former semantics meaning the message was either successfully delivered or the delivery failed. Functionality like IPC timeouts and cancellation as well as flow control must be implemented by higher level protocols.

During the asynchronous delivery, the message must be stored somewhere. Formerly the sending thread was blocked and the message data was stored in the resources of this thread (e.g. fields in the TCB). However, as the thread is now executing when the message data is still needed this is not possible anymore. The message could be stored within the resources of a proxy thread or within the kernel. If the message is stored within the proxy it must either provide a thread for every pending message, so the TCB fields could be used, or the proxy must provide explicit memory. If the message is stored within the kernel a resource management must be implemented for the message storages. A message storage could be some kind of virtual (passive) senders representing a message request just like the virtual senders that represent IRQs.

7.1.4. Interrupts

A naïve approach of managing interrupts is to manage nothing at all. All processors accept external interrupts. If a thread is attached to the interrupt on the processor that caught it, the interrupt IPC message is sent to this thread, but if there was no thread attached the interrupt is simply lost. So the user space is responsible for programming the IO-APIC to deliver the interrupts only to processors on which an attached thread runs. This gives to full control over the external interrupt system to the user space.

Another simple approach is that there is a dedicated processor for interrupt handling. Only on this processor threads can receive interrupt messages. This would be as simple as the naïve approach but not so flexible.

A more complicated way is that the kernel delivers the interrupt message to the nearest attached thread. This means if a thread is locally attached the message is sent to this one otherwise the interrupt is forwarded to a processor that has a thread attached. To avoid the forwarding of interrupts the kernel might program the IO-APIC itself, so interrupts are only delivered to processors that have an attached thread.

7.1.5. New Cross Processor Operations

Possible candidates for cross processor operations are thread placement/migration, task management, cross processor signalling, or cross processor IPC. For every operation that works across processor boundaries additional infrastructure or synchronization mechanisms are needed, which automatically increases the complexity of the kernel.

Cross Processor Communication

A newly introduced operation could be a cross processor IPC. If proxies are used this this discussion relates to the communicatin between the proxies. This operation can be synchronous or asynchronous. For both variants about the same complexity is expected. For the synchronous one the cross processor synchronization mechanism must be implemented and for the asynchronous one an in-kernel resource management is needed.

IPI as New Abstraction

In combination with proxies no direct cross processor message passing needed. For the synchronization among the proxy threads a signalling mechanism like inter processor interrupts is sufficient because the transfer of data could be done using shared memory. Because receiving IPIs is similiar to receiving other

external interrupts a thread (esp. proxy thread) can attach to an IPI. To send IPIs to other processors a new operation must be introduced. This operation could be mapped to virtual threads, too. So sending a special message to a virtual IPI thread leads to sending an IPI to another processor.

7.2. General Global Model

The *general global model* intends to provide complete transparency for multiprocessor unaware applications. The mechanisms and abstractions for IPC, address space management, and thread management have a system global scope. However, the scheduling is done processor locally and thread migration is introduced as a new cross processor operation in order to move threads from one processor to another.

The choice for the processor local scheduling model contradicts the requirement for transparency but as already mentioned a transparent scheduling model would incur an infeasible performance penalty.

On the first glance this is the desired user model for the multiprocessor supporting L4 microkernel. The approaches of Völp [29], Uhlig [28], and Peter [25] can be classified as variations of the *general global model*.

However, the implementation of the *general global model* can become more complex as shown in 9.

7.3. Split Model

This section discusses the parameters of an alternative to the *general global model*. As a working hypothesis it is assumed that the multiprocessor port of L^4Linux only needs a signalling mechanism and memory that is accessible from every processor it runs on. This seems plausible as multiprocessor systems provide nothing more than these to things for native Linux. Further it is assumed that the L^4Linux design does not destroy the multiprocessor design characteristics of standard Linux.

The idea behind the *split model* is that an uniprocessor kernel is already available, or at least is simpler to develop. This uniprocessor kernel is partially duplicated so that one duplicate runs on each processor. This brings the added kernel complexity to a minimum as the duplicates work nearly independent. Also, this will automatically limit the scope of all existing operations and the scheduling to be processor local. Although the memory mapping is only processor local the physical (user) memory is shared among the processors. The only cross processor operation will be register-value short IPC; thread migration will not be supported.

As scheduling is processor local and no migration is supported no extra synchronization for concurrent migration and other operations is needed and therefore the kernel complexity is not needed to be increased for this. Further as other operations such as execution state manipulation (`lthread_ex_regs`) and task creation/deletion are processor local no additional infrastructure is needed. So the partially duplication leads to one mapping database instance per processor and therefore a complex cross processor synchronization of its datastructures is not needed. Because the duplications are almost independent the realtime properties are kept in the processor local scope.

A cross processor infrastructure is needed for the cross processor IPC. This results in additional code in the kernel. However, this new code is virtually independent from the existing one and therefore such extensions are uninstrusive.

The *split model* totally exposes the multiprocessor view to user land. The operation of mp-non-aware applications will be restricted by the processor boundaries. As this model provides shared memory and a cross processor signalling mechanism it is expected that L^4Linux can cope well with it according to the working hypothesis. A more detailed discussion concerning the usability of this model follows in section 8.

8. Multiprocessor User Environment

This sections discusses how well the different parameters of the split model work with the intended work load scenario of L[4]Linux [8, 19, 20] and L4Env [1] The starting point is a user model that is most convenient for the kernel. This user model might be changed if it is too inflexible for the targeted work load.

The assumed model is a split model providing shared physical memory, split virtual memory with no cross processor mechanisms for virtual memory management. Further the model will provide no cross processor mechanisms for thread management, and there is a synchronous cross processor IPC mechanism.

8.1. L4env

Applications must be made aware of the multiprocessor system because the cross processor boundaries are visible now.

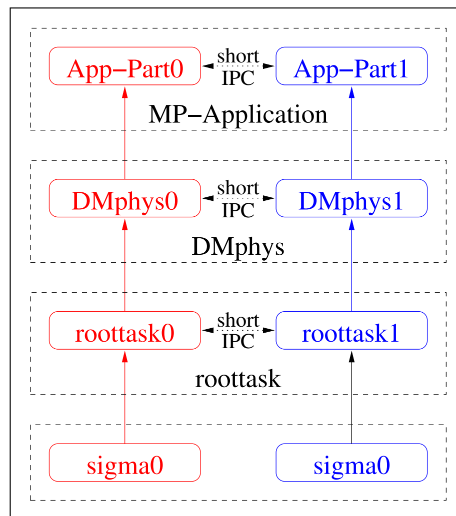


Figure 3: Setup for a L4env Multiprocessor Application

Figure 3 shows a scenario that could be built on top of the very rudimentary *split model*. Each multiprocessor server and application consists of multiple tasks, one for each used processor. All tasks of a single application (resp. server) are congruent, i.e. all tasks contain the same virtual to physical memory mapping. This simplifies the access to shared data by the applications. Thus the mapping hierarchies should be congruent at least for the L4env servers and multiprocessor applications.

Three basic design patterns are available for servers. These three different approaches could be applied in combination so for each server functionality an appropriate pattern can be chosen. The first, a very simple one, is to instantiate independent servers on each processor. This, of course, is only possible if the provided service is processor local and independent of the same service running on other processors. As such a server instance is constraint to a single processor it cannot exploit the internal parallelism. However, this is a good fit for servers that manage processor local resources, such as tasks that are managed by the *task_server*. So for cross processor task creation the startup server, i.e. *exec*, must ensure that the *task_server* on the destination processor is contacted.

The second pattern is called server stubs. A server stub is a kind of proxy thread within the server. On each processor other than the first one, runs a stub thread. The stub threads receive local requests and forward them to the main server thread. They also execute operations on behalf of the main thread if this

operation is not cross processor capable. Server stubs cannot exploit internal parallelism. However, they can provide services like supplying memory, starting tasks on multiple processors so applications that use multiple processors can be constructed on top of them. The server stub methodology is a good fit if an existing multiprocessor unaware service shall be adapted to an multiprocessor system, in particular if the service is not performance critical.

The third design pattern is a multi-threaded server. On each processor a service thread is available and handles only local requests. The access to data structures that are not processor local have to be synchronized. Multi-threaded servers offer high functionality at the cost of high development costs.

σ_0 and Roottask

σ_0 gives all its memory to roottask so it is only during the startup of relevance. During initialization it prepares internal data structures. These data structures as well as memory regions like kernel memory are registered in order to prevent them to be mapped to other applications. As each task of σ_0 has to know about the data structures of the other ones a global registry is needed which also demands some kind of synchronization. A synchronized cross processor signalling would provide a simple synchronization mechanism. As the cross processor IPC can be used for this the model is sufficient for this purpose.

The services provided by the roottask are not compute constrained which makes it a good fit for the server stub design. The cross processor task creation can be forwarded to the server stub that is running on the target processor. σ_0 protocol requests can be forwarded to the main thread of the roottask server. For synchronization and communication between the main thread and the stub threads a cross processor IPC operation that is synchronous and transfers at least a single machine word is desirable in order to make user level synchronization simpler and to transmit a pointer value to a data structure that contains the real but far longer message.

Because there is no memory mapping and no task management across the processor boundaries possible σ_0 and roottask instances have to be established on each processor by the kernel upon startup.

L4env Servers

For the multiprocessor port of L4env servers the server stub design pattern as well as the multi-threaded server design pattern could be used. Similiar to roottask it is preferred to have a synchronous cross processor IPC to transfer at least a machine word.

Application Startup

For the application startup the libraries needed to load an application could be modified to allocate new tasks on other processors, to register these tasks to the different dataspace managers, and to signal the task server that these tasks were created. On each processor a thread is needed that does the processor local initialization. Another library that is to be modified is the region manager that is responsible to resolve pagefaults for applications threads. This library has to provide a pager thread in each task of the application. The libraries can be organized using server stubs as well as using a multi threaded library approach.

8.2. L^4Linux

Figure 4 shows a multiprocessor setup of L^4Linux. As multiprocessor L4env application it has congruent tasks on each processor. This gives the L^4Linux server a memory view similiar to the view that a native Linux has on multiprocessor systems.

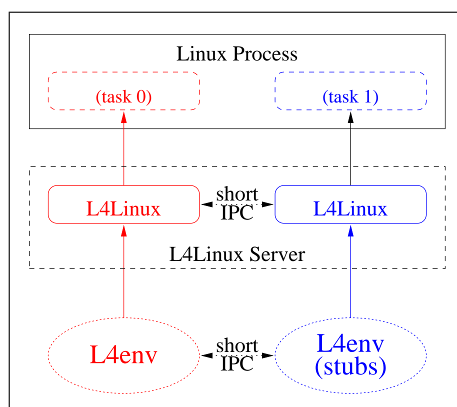


Figure 4: L4Linux Multiprocessor Setup

Synchronization

The first synchronization type used by native linux is lock based synchronization. Memory based locks will work automatically.

Linux also uses IPIs for cross processor synchronization. Receiving an IPI is just a special form of receiving an interrupt. Therefore for incoming IPI interrupt threads must be provided. The IPI receiving threads are handled just like interrupt threads in L4Linux. But instead of attaching to an interrupt it just awaits a signalling message.

Memory Model

Translations of virtual memory addresses to physical ones are defined in page tables and are cached in the TLB. When a page table entry is invalidated or changed all cached instances of it have to be found and invalidated. Since TLBs cannot be invalidated remotely each processor has to be notified to purge its own TLB.

Like TLBs, L4 address spaces cache L4Linux' address translations. In fact L4Linux can handle them as if they were software loaded TLBs: If a mapping is not yet available in the L4 task, L4Linux does a software page table walk to find the correct translation and inserts the mapping into the L4 task. Furthermore, if an entry in the Linux page table is changed all mappings in L4 tasks that represent this translation must be *flushed*. As this is the same algorithm as the TLB shootdown in Linux the existing infrastructure can be reused.

Threads

In L4Linux there is one single L4 thread per Linux process. This L4 thread will be multiplexed to the different Linux threads belonging to the Linux process. The multiplexing is done by the L4Linux server by extracting and setting thread's execution state via `thread_ex_regs` accordingly.

As L4Linux stores the state of not running Linux threads within the L4Linux server these states can be read on all processors. Migration mainly means for Linux to schedule the thread on another processor. Linux must not care for migration of other resources needed by the thread. So this could be done by scheduling the thread on another processor, i.e. to load the state of the Linux thread into an L4 thread on another processor. So the missing migration in the microkernel model does not prevent Linux to implement it on top of it for Linux threads.

In L^{4}Linux a single L4 task id is mapped to a single linux process id. For the multiprocessor port this is no longer possible because there are multiple L4 tasks per process. The mapping must be changed so that the processor id and the process id together determine the L4 task.

Linux Drivers

Drivers for DROPS components or other L4env servers have to be able to cope with the situation that the actual server might not run on the same processor. It might be necessary to have a linux kernel thread at the processor where the corresponding server is running and requests have to be forwarded to this kernel thread.

Part IV.

Design

9. General Global Model vs. Split Model

After the kernel complexity that is needed to implement the *general global model* (7.2) and to implement the *split model* (7.3) is discussed, a short motivation for the choice of an user model based on the *split model* is given.

Expected Complexity of the General Global Model

The *general global model* is assumed to provide address spaces that span across processor boundaries, threads that can migrate from one processor to another. Further, it is assumed that processor local operations are much more frequent that cross processor ones. Thus the local operations shall be optimized.

Operations are executed on threads. If the operation normally operates on tasks, the task will be identified with its first thread. If the thread of interest currently resides on the processor of the initiator of the request the operation can be executed directly with interrupts disabled. If not a request must be sent to the other processor. This request is put into a request queue that belongs to the target thread. However, the processor of the target thread must also be notified. Therefore another request, which only contains a reference to the target thread, is enqueued to a queue that is associated with the target processor and finally an IPI must be sent there. This two-staged request sending is necessary to implement a simpler thread migration: The pending requests are still associated with the migrating thread. The new hosting processor is responsible for the handling of these requests, which are still found in the queue of this thread. There is a request for thread migration just as there are some for the IPC rendezvous, IPC cancellation, or `lthread_ex_regs`.

The mapping database becomes a globally shared resource and needs appropriate locks. After an unmap stale TLB entries must be invalidated. This could either be done with explicit shootdowns or more gently with RCU. The required preemptibility also effects the shared mapping database and thus could lead to very long, but not infinite, operation latencies. As a result realtime operations must not access the mapping database. Actually this is also true today as the number of map items to be removed is unbounded.

Priority donation is also possible if the donator and the donatee reside on the same processor. If either one migrates the donation must be cancelled.

As an alternative it is possible to use system global locks. The weakness of this approach is that getting a global lock even impairs performance of processor local operations. However, locks could be refined to have a smaller scope and thus smaller contention.

Expected Complexity of the Split Model

In the *split model* it is assumed that address spaces do not span across processor boundaries and cannot migrate. Thread migration is also not supported.

The existing kernel infrastructure is simply duplicated and used only processor locally. Further a small cross processor infrastructure has to be supplemented to support cross processor operations.

The model rules out thread migration and system wide TLB invalidation as issues since it does neither provide the thread migration nor address spaces that span across processor boundaries. The mapping

database does not become a globally shared datastructure since the mapping hierarchy is bound to the local processor.

Benifits of the Split Model

The thread lock need not to be changed. It can still be used for IPC and thread synchronization without impairing the performance.

The message box system need not to be two-staged, where one request is sent to the target thread and one to the target processor. This is because there is no thread migration and thus there is no need to migrate pending requests along with the thread they belong to.

As the mapping database is only processor local, there is no need to protect it with a global lock. Further, there is no need for TLB invalidation since there is no address space that is spanning across processor boundaries.

10. Partitioned User Model

This section describes the user model that was chosen because of the results derived in the previous part of this document. It will be a summary and refinement of the section 7.3.

10.1. Memory and Tasks

The model provides only processor local task management. Memory mappings are also restricted to the local processor. Further, the chief of a task must be located on the same processor. This implicitly restricts the task creation and killing to the local processor because these operations are only allowed to be executed by the chief of the manipulated task. Finally, tasks are bound to the local processor and do not span across processor boundaries.

A single continous task id range is assigned to the processors so that each gets about the same number of tasks. Because the task id is system global a task is visible on every partition.

Because the mapping hierarchies are processor local for each processor a root address space σ_0 is needed, which will be the origin of all local mappings. The physical memory is not partitioned and it is mapped one-to-one to the σ_0 tasks. So it is mapped congruently on each partition.

Because the roottask is the initial chief of all tasks an instance of it must be running on every processor. This means the roottask instance is the initial chief of all tasks that are located on the same partition.

All initial tasks that are needed for σ_0 and roottask are created by the kernel automatically at startup.

10.2. Threads

In L4v2, the threads are bound statically to tasks. Therefore they are also statically bound to a processor like the tasks. So no migration is supported. Like tasks threads also have a system global identifier.

The already in Fiasco existing scheduling model, which provides realtime functionality like quality assuring scheduling (QAS), is restricted to a processor local scope. Scheduling parameters like the thread priority can only be set by threads that belong to the same processor. Further, time-slice donation only is

supported for the processor local IPC. Cross processor IPC is handled by the scheduling model like other external events, such as interrupts. This means that it only results in blocking or unblocking of threads which includes a possible need for rescheduling.

The `lthread_ex_regs` mechanism is restricted to the local processor since the discussion in section 8 showed that the environment can be modelled without it. Further, this decission saves the additional infrastructure needed for this operation in the kernel.

10.3. Communication

To avoid a complex adaption layer (e.g. proxy threads) the communication between application threads of different processors is directly. As the thread ids are global the threads can represent communication end points for the cross processor communication; this is similar to the original L4v2.

The cross processor IPC is restricted to the synchronous transfer of register messages. This avoids expensive resource management for message buffering.

10.4. Interrupts

The best solution is the naïve approach where the interrupt routing and acceptance is entirely controlled from user space and the processor handles every incoming interrupt only locally. However, this whould mean that the user land has to be extended in order to program the IO-APIC. That is outside the scope of this work. The other simple possibility was to accept interrupts only on the first processor. As the kernel implementation can be changed very quickly between these to variants the prototype just uses the model of accepting interrupts only on the first processor.

10.5. Cross Processor Operation

The only cross processor operation provided by this model is the register-only short IPC. The operation uses the same kernel interface as the processor local IPC operations. The cross processor operation is distinguished from the processor local operation by the thread id of the destination thread.

11. Kernel Design

In this section the basic design idea is illustrated first. Afterwards specific design aspects are discussed.

As shown in *figure 5*, the general idea of the design is to partially duplicate an already existing uniprocessor kernel. The duplicated parts of the kernel work virtually independently from the execution on other processors. For the cross processor operation a new infrastructure is introduced that is widely independent from the rest of the kernel. This infrastructure is the only contact point between the duplicated kernel parts. So for most parts of the duplicates no cross processor synchronization is needed. Further, the realtime properties are still held processor locally if the new cross processor infrastructure complies with the realtime framework.

In 11.1, the duplication of variables belonging to the duplicated parts of the kernel is discussed. The different synchronization mechanisms used by the kernel are discussed in 11.2. In section 11.3 the design of the new cross processor IPC operation is presented. Finally, the other parts of the cross processor infrastructure are discussed in 11.4.

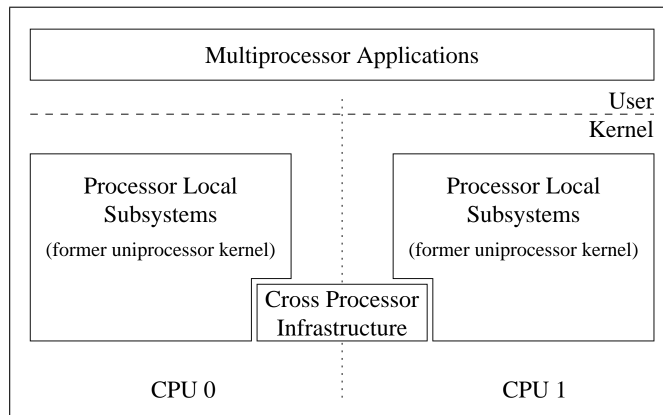


Figure 5: Partial Duplication of the Uniprocessor Kernel

11.1. Data Ownership and Data Access

The duplication of kernel subsystems mainly addresses the duplication and separation of kernel data objects belonging to these subsystems. However, there are also kernel data objects that are used by the cross processor infrastructure and thus are not duplicated.

This section will describe the duplication of variables that are global or static in the single processor kernel and variables that are used for the cross processor infrastructure, the usage of dynamically allocated kernel objects, and as a special case of dynamic kernel objects the access to thread control blocks.

11.1.1. Processor-Local Global and Static Variables

Some synchronization that is used in single processor kernels works only processor locally in multiprocessor systems (see 11.2.1). Thus, if a variable is not duplicated, it is necessary that the synchronization primitive that is used to protect the variable is exchanged. If the variable is used by a duplicated subsystem, it must be duplicated, too.

This section describes three different ways that can be used access global and static variables.

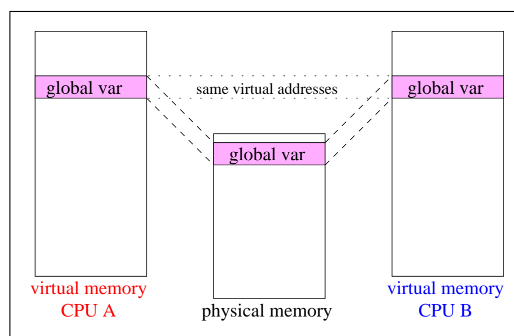


Figure 6: System Global Variables

The first option is used for cross processor infrastructure because the variable is not duplicated. Thus subsystems that use this access scheme do not become independent. The variable is shared among the processors as shown in *figure 6*. The access to it is protected by a multiprocessor aware synchronization

scheme. This access scheme introduces overhead for the locking primitive and overhead for the cache migration of the accessed variable.

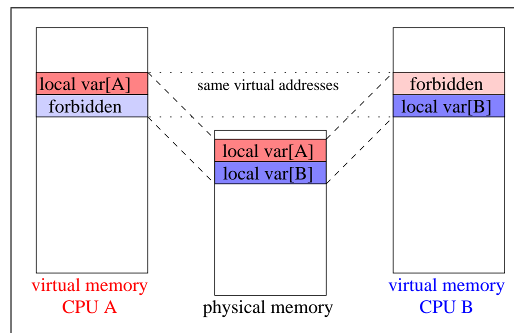


Figure 7: Processor Local Variables by Array Indexing

To make the subsystems on different processors independent the variable could be instantiated on each processor. This means each processor works with its own copy. *Figure 7* shows a variant where an array is used instead of a single variable. Each processor uses another index to access its own variable. So no synchronization is needed anymore but an additional indexing operation is introduced. There are about 700 global and static variables in the Fiasco kernel. All occurences of these variables in the source code had to be adapted to the new scheme. This would result in a very instrusive code modification.

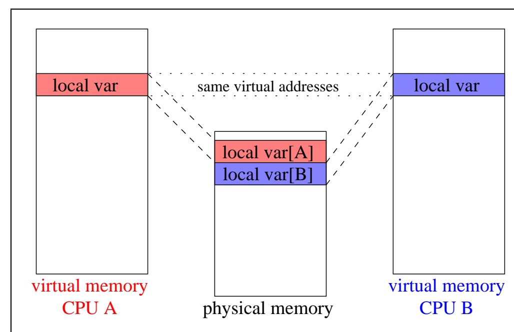


Figure 8: Processor Local Variables by Memory Mapping

As shown in *Figure 8*, another option is to have a per processor kernel address space where the same range of virtual addresses on different processors is backed by different physical frames. Basically this changes the explicit indexing done in the program code to an implicit indexing done by the MMU of the processor. The overhead of the explicit indexing is eliminated.

The third option is the general choice for all global and static variables. However, as the first option is needed for the cross processor infrastructure it is possible to explicitly tag variables that shall be located in a virtual memory region that is backed by the same physical memory on all processors.

11.1.2. Allocated Pages

Some kernel subsystems, e.g. slab allocators, the mapping database, or the paging, use dynamically allocated memory. Pages for kernel usage are allocated from a shared memory pool. The allocated pages are assigned to the processor that requested them - the processor becomes the owner of the page. A processor is only allowed to access pages it owns. The ownership is ended by giving the page back to the memory

pool. This conforms the access convention on single processor systems, so this restriction needs no further implementation efforts. However, there are two exceptions to this convention.

Page ownership can be transfered between processors. The former owner then is not allowed to access or to return the page anymore. The new owner is now responsible for freeing the page when it is not needed anymore. The message box system that will be described in section 11.2.3 interrupts the local execution in order to handle a remote request. If during this request handling dynamically allocated memory is needed, the interruption time can be reduced if the pages are allocated beforehand and passed to the processor that handles the request.

Another exception to the access convention described above is made for pages that are used by cross processor infrastructure like cross processor IPC or the message box system. This means these pages contain data that is accessed by multiple processors. This violates the convention that only the owner accesses a page. Therefore it must be ensured that no processor uses these pages anymore before releasing them. However, as these pages are never released this problem will not occur here.

11.1.3. Thread Control Blocks

It is not possible to apply the concept of data separation to TCBs because they are used by the cross processor IPC. Therefore, it is necessary that TCBs can be accessed even by processors that do not host the corresponding thread. This sections describes the access rules for TCBs.

Thread control blocks (TCBs) are located in a designated memory area and are globally visible. Such TCB is managed by the processor to which the thread belongs. Only this processor is allowed to write the TCB. This is like the approach that uses arrays for the data separation. But this time no extra overhead for the indexing is introduces since the indexing is already present in the single processor kernel. However, remote TCBs can be savely read using atomic read instructions.

The only exception is made for two pointers that are used for the waiting sender queue. By convention only that processor where the receiver is located is allowed to manage this queue since it belongs to the receiver. The sender can only be enqueued in a single waiting queue. So if it is necessary to enqueue it the processor of the receiver is allowed to access these pointers until the sender is dequeued again.

11.2. Synchronization

In 11.2.1, synchronization schemes are discussed that are used in the single processor Fiasco kernel. A cross processor spinlock is introduced in 11.2.2. Finally in 11.2.3, the design of the message box system is presented.

11.2.1. Traditional Single Processor Synchronization

This section discusses synchronization mechanisms that are used in the single processor Fiasco kernel disabling interrupts, atomic machine instructions, and helping locks.

Disabling Interrupts

The assumption on single processor systems is that disabling the interrupt makes the thread temporarily non-preemptible provided that no other exceptions cause preemption and hence no other code can execute concurrently. But on multiple processors systems there are still threads running on other processors that can interfer. The new correct assumption on multiple processor systems is that there is no local interference

possible. It would be to expensive to enforce that no other thread is running in the system because all processors expect one are idling. If this synchronization shall not be used to access threads, a relaxed semantic is sufficient. It would be enough to disable interrupts locally and to use a cross processor spinlock. Because this synchronization method is used fequently, lock contention is very probable. Further, the frequent use leads to frequent cache migrations of the spinlock. So another, and more complex, locking primitive had to be used.

Atomic Instructions

For this synchronization scheme the processor provides instructions that fetch data from memory, modify it, and store the result back to memory. As these instructions cannot be interrupted they seem to be atomic. This is no longer true in multiprocessor systems because the two subsequent memory accesses can be observed by other processors. So if the memory accesses are serialized in a way that between the read and write parts of such instruction another write to the same memory location is executed by another processor its written value will be lost. To circumvent this problem the processors provide the low level synchronization of bus locking. While an instruction is executed under the protection of the bus lock, the accessed variable cannot be modified by other processors. Instructions that use bus locking need more time to execute than their counterparts without bus locking as it was measured by Reusner [26].

Helping Lock

Helping locks implement wait-free synchronization [11] and are currently designed for single processor systems. If a thread that holds the lock, i.e. is lock owner, is preempted and another thread wants to take the lock, this other thread helps the current lock owner until it releases the lock. Helping is done by switching to the execution context of the current lock owner. While the helping takes place, the lock owner consumes time from the time-slice of the helping thread.

Since time slices are bound to a specific processor, the lock owner has to be executed on another processor while helping cross processor [12]. This results in, at least, two migrations, one to the processor of the helper and one back. Further migrations can occur if the helper thread is preempted and yet another thread wants to help. This needs a more complex implementation of the lock because it must be ensured that the thread that is to be migrated is not running at the moment and has not yet released the lock. Further, if multiple helpers are available all kind of interaction must be considered: A thread must be able to determine whether helping should take place because the lock owner could already be executing or another thread could already be helping. This synchronization can be achieved by atomic state modifications or an extensive use of notification IPIs. Both variants incur a severe runtime overhead. Further, if a helper, or the lock owner itself, is preempted and other helpers are available, one of them should be activated in order to continue the helping.

While holding a helping lock some kernel parts also use interrupt disabling as further synchronization method. A thread can migrate to another processor at anytime while it holds the helping lock. This makes it impossible to use processor local variants of the synchronization within helping-lock-protected code parts.

Conclusion

All three synchronization types still work in multiple processors systems if only used for synchronization among threads running on the same processor: If the interrupts are disabled no other local thread can

preempt the currently executing one. Machine instructions cannot be interrupted by other local instructions so if applied to processor local datastructures the instructions still appear atomic. And if it is ensured that only local threads try to help when aquiring the helping lock no temporary thread migration is needed.

So these synchronization schemes are suitable for subsystems that work entirely processor-local. For subsystems that need cross processor synchronization other means are needed. As the cross processor synchronization is more expensive processor local synchronization should be used if possible.

11.2.2. Spinlock

A very simple synchronization primitive is the spin lock. The processor waits for the lock to be free in a busy waiting loop. When the lock is free it tries to aquire the lock. If this fails it reenters the waiting loop otherwise it enters the critical section.

If the code enclosing the spinlock is preemptible the thread must be preemptible while waiting for the lock. But at the moment the lock was aquired interrupts must be turned off. If interrupts were allowed, the thread could be preempted and it might take a long term until the lock is released. It also bears the chance of priority inversion. The probability that another thread is waiting for the spin lock is increased and thus the spin lock would become a bottleneck. This means lock acquisition and disabling interrupts must be one atomic operation.

However this lock should be used for short and infrequently executed critical sections. Further the critical sections must be left within predictable time so the interrupt latency, which is the time needed for a higher priority thread to preempt the currently executing thread, is still bound. If these preconditions are not met other means of synchronization like the message box system, which is explained in the following section, should be used.

11.2.3. Message Box System

Data structures like TCBs are used by both the processor local and the cross processor operations. Concurrent accesses by multiple processors need to be properly synchronized. The assumption that a huge majority of the executed operations are processor local implies that there would be much overhead for a functionality that is rarely needed if a cross processor synchronization scheme was applied. In order to make cross processor operations possible even if the synchronization for these data structures is processor-local, a request to execute a critical section can be sent to the processor where the synchronization is actually done. This processor can always be determined due to the static placement of the object.

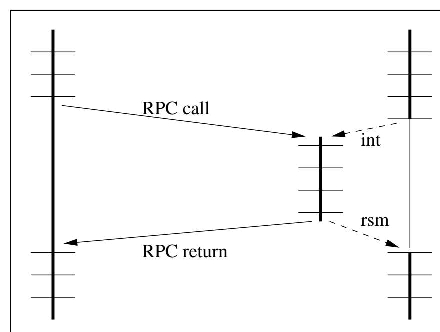


Figure 9: Remote Procedure Call of the Message Box System

The message box system is a kind of a remote procedure call infrastructure. As shown in figure 9, a thread

can send a request message to another processor. The target processor will execute the request handler asynchronously to local threads.

For simplicity a processor can have only a single outstanding request and blocks until it is finished. If the execution were continued with another thread, this thread could issue an `lthread_ex_regs` that cancels the system call of the thread that is waiting for the response. After it returns the effects must already be visible to the user space. If this operation is applied to a thread that had started an IPC before, no other thread must receive the sent message after the `lthread_ex_regs` returned. A simple way to achieve this is that `lthread_ex_regs` just waits until the effect took place. For the IPC operation that also uses requests it must be waited until the request was handled. Therefore, more than one thread would have to wait for the request to be handled and the code path of `lthread_ex_regs` had to be modified, too.

To avoid deadlocks the processor must handle incoming requests while waiting for an response. Otherwise the handling of incoming requests depend on the handling of the outgoing request and crossing messages could produce a cyclic waiting dependency.

Nested request must not be issued by a request handler because it can be executed while the processor is waiting for an own outstanding request. The system is designed in a way that nested requests are never used.

The critical section executed in the request handlers shall be short in order to minimize the interruption time of the processor where the request is handled and the blocking time of the processor that sent the request.

The order in which an incoming request and the awaited response are handled has to be the same one in which they were sent. This property is needed for the atomic transition from send to receive in the cross processor IPC path.

11.3. Cross Processor IPC Path

For the cross processor infrastructure that implements the cross processor IPC two main possibilities are available.

The first option is to change the already existing IPC path in a way that the operations that need cross processor synchronization are sent as request to the target processor. This means for a naïve implementation that five messages are needed if the receiver is ready to receive and ten if the receiver is not ready. With some optimizations these numbers can be improved to three and eight, respectively. If the IPC partner is not yet at the rendezvous, the IPC path needs to aquire the thread lock, which is a helping lock. Helping locks are only a processor local synchronization primitive (see section 11.2.1) but the cross processor IPC path would need a lock that works processor locally and also works across processor boundaries. Therefore the helping lock could be extended to allow a remote thread to hold the lock without the possibility to help the lock owner cross processor.

The second possibility is that a dedicated code path for the cross processor IPC send bypasses the processor local IPC send. The receiving path must cooperate with both the cross processor and the processor local send path. A new IPC path can be designed with a single message if the receiver is waiting for a sender and two messages if the receiver is not waiting.

11.3.1. Requirements

The dedicated cross processor IPC path must be integrated with the current IPC path. The atomic transition between the send part and the receive part of the system call must be guaranteed even if the send operation was a cross processor send. Another requirement is that the cross processor send path works with the

current receive path. This is needed because for an open receive the thread does not know whether the sender will be located at another processor or be a local thread. Further, the cross processor operation must use the same binary kernel interface as the processor local one.

It is an acceptable compromise between kernel complexity and user functionality if the cross processor IPC path only implements register-only short IPC. It also should support cancellations and timeouts. And finally, the performance impact on processor local IPC should be as little as possible.

11.3.2. Design Aspects

Cross Processor Synchronization

In Fiasco, the processor local IPC path is partly non-preemptible. The non-preemptible part of the IPC path is the one that implements register-only short IPC. It uses interrupt disabling as synchronization primitive.

The cross processor IPC will use the message box system as infrastructure for cross processor communication and synchronization. The request handler of the message box system also uses interrupt disabling thus remote operations concerning the IPC path are automaticly synchronized with the IPC path.

Waiting Queue Access

The manipulation of the sender waiting queue is done by the processor of the receiver. This processor gets the permission to access the corresponding fields in the TCB when the message is tried to sent.

This permission is implicitly revoked when the send operation finished, regardless whether the message was finally delivered, the delivery failed, or was cancelled.

Because a sender can send a message only to a single receiver at the same time only a single processor will access the TCB fields belonging to the waiting queue.

Reading of Remote TCBs

The receiver can read the register message of the sender, which is part of the TCB. Because the message contains two machine words the sender must ensure that these will not change during the message transfer. This constraint is implicitly met because the register message is not modified while waiting for the delivery of the IPC message. It is only modified when continuing with the receive operation or returning to user space. But in both cases the send operation is already finished and the register message is not accessed by the receiver anymore.

11.3.3. Cross Processor IPC States

The states of the cross processor IPC *send* operation are illustrated in *figure 10*. In the *prepare IPC* state the send operation and an optional receive operation are initialized. The receive operation must be initialized here because of a potential atomic transition to the *receiving* state.

When sending the *IPC send request*, which is part of the cross processor rendezvous, the state changes to *wait for reply*. This state is left after the request was handled by the remote processor. If the send operation was already successful, i.e. the message could be delivered to the receiver during the request handling, the send operation is already finished. If the receiver was not ready to receive any messages but a timeout of

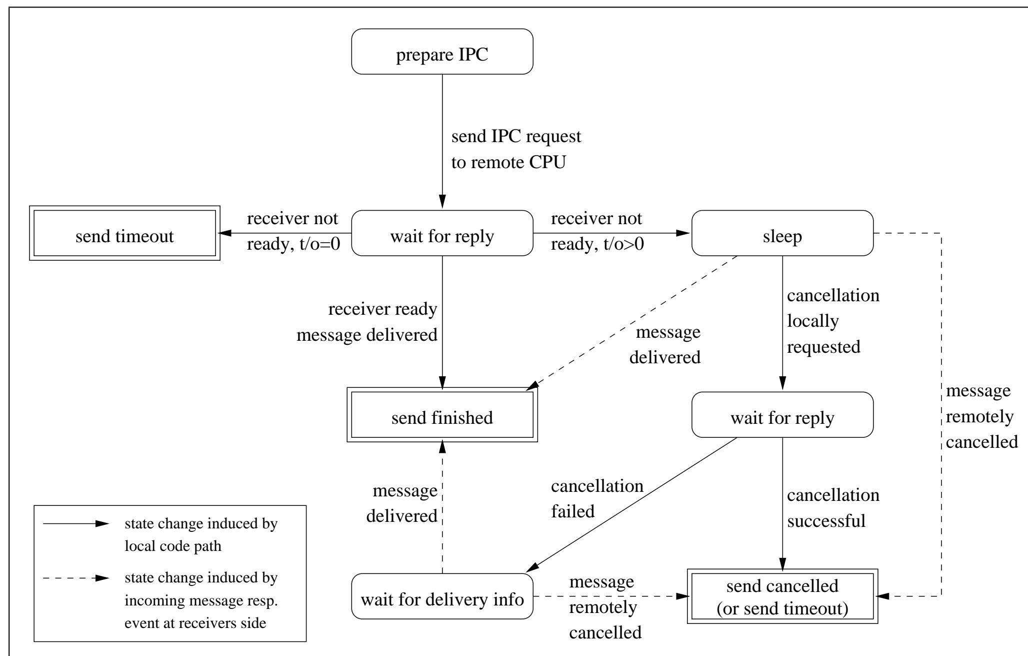


Figure 10: State Diagram of Cross Processor Send Operation

zero was given the operation failed due to a timeout. If there was a timeout other than zero and the receiver was not ready the sender was enqueued to the waiting sender list of the receiver during the request handling and the sender *sleep*s.

If the timeout is neither infinite nor zero, the sender initializes the timeout. There are three reasons why the sender can wake up: The message was eventually delivered and thus the receiver sent a *sender wake up request* back to the processor of the sender. A cancellation of the IPC operation is locally requested by an expired timeout or because of an `lthread_ex_regs`. A third reason could be that the receiver thread was killed and so a remote *cancellation request* arrived.

If the sender woke up because the message was delivered, it finds its thread state reset or receiving depending on whether a receive operation is pending.

If the IPC operation was remotely cancelled, by the termination of the destination thread, the corresponding flag is set in the thread state. In this case the sender was already dequeued from the sender waiting list. The appropriate failure code is returned.

If the sender woke up because the operation should be cancelled due to a local reason (timeout or local cancellation) the sender sends a cancellation request to the remote processor and enters the second *wait for reply* state. The remote processor tries to cancel the operation. This results in a race between the message delivery and the cancellation request. If the cancellation is early enough, i.e. the message was not yet delivered, the response to the *cancellation request* is that it succeeded and a timeout or cancellation error can be returned. If the message had already been delivered when the remote processor handled the *cancellation request*, a failure response is returned. The cancellation only fails if the *cancellation* request and the *wakeup request* were sent simultaneously. So the sender must wait for the *wakeup request*. In fact, the awaited message is already in the message box because delivering the IPC message and sending the request to wake the sender is done atomically. After handling the message, it will be proceeded as if this message woke the sender.

11.3.4. Atomic Transition from Send to Receive Stage

The transition from send to receive stage of an IPC system call must be atomic in order to enable user level servers to send a message to their clients with the timeout of zero: The client that sent its request to the server is called original sender in this section. When this client sends a request to a server it must be ready immediately after the request was accepted by the server so it can to receive the answer. If this were not the case and the server replies with an timeout of zero the server answer could not be delivered since the client were not ready to receive.

In the cross processor IPC send path there are two code paths where the transition from send to receive happens. The first one is taken if the response to an *IPC send request* states that the message was successfully transfered. The second one is used if the sender was woken by the receiver that completed the message transfer.

For the first path, it is necessary that the awaited response is handled before any request that might be sent after the response: When the message is delivered to the receiver while handling an *IPC send request*, the receiver is activated immediately and thus can invoke the next IPC operation. This operation could include an *IPC send request* that is sent back to the original sender. If requests could overtake responses, it would be possible that the second *IPC send request* is handled while the original sender is still waiting for the response and thus is not yet receiving. This would contradict the demand of the atomic transition. So the handling of the second request must be postponed until the original sender is in the receive state.

In the second path the sender that was woken by the receiver is not scheduled immediately. However, the receiver does not wait for the sender to be scheduled and thus could already continue with another IPC operation in order to send a message back to the original sender. The *IPC send request* initiated for this purpose must not find the original sender still being in its sending state. To achieve this the thread could be put into the receiving state while handling the *wakeup request*. But as the request is handled outside the IPC path some information needed to setup the receive operation is not available. Therefore the optional receive operation is prepared in the IPC path before the send operation is started. So when handling the request only the thread state needs to be changed. This procedure was abolished with the switch to the non-preemptible IPC path but is revived in the multiprocessor version. But still the message handler needs the information whether a receive operation follows the send or not. Therefore an additional flag as part of the thread state was introduced.

11.4. Miscellaneous Cross Processor Functionality

This section mentions trivial cross processor functionality of the kernel.

11.4.1. Kernel Console

The kernel console is only used for development purposes. In production systems output to the console is managed by an user space server. Therefore a more pragmatic solution can be used here. The state of the video hardware which is needed to be known on every processor is the cursor position. If threads on different processors are writing all the output is seen on the screen. If it is interlaced no characters will be lost.

The access to the cursor position variables, the serial interface, and the text console frame buffer is protected by a spinlock.

11.4.2. Low Level Memory Allocator

The low level memory allocator is the free memory management of the kernel. In the uniprocessor kernel, it is implemented as sorted list of free memory ranges that are not adjacent. This allocator only works for datastructures that are at least as large as a page and their size is a power of two: e.g. page tables and directories. For all other data structures slab allocators are used. The slab allocatores cache pages allocated via this low level allocation. Because the low allocator is only used during task and thread creation operations and the slab allocators that cache the allocated memory, even in the cross processor kernel no contention is expected.

The low level memory allocator is shared among the processors. If it was not shared a balancing mechanism had to be implemented so memory does not run out on a single processor. As there is no contention expected and the allocator does not ensure any execution times it seems to be sufficient to protect the low level allocator with a spin lock as described in section 11.2.2.

However, as the spinlock disables interrupts, some modifications to the allocator must be made to meet the realtime requirements of bounded interrupt latency. Releasing memory has originally a time complexity of $\Theta(n)$ because the organisation as sorted singly-linked list. The release operation consists of putting the new item to the correct place in the list followed by an iteration over the whole list and merger of adjacent ranges. The allocation of single pages is done in $O(1)$. Because the list uses the *first fit* strategy, it always takes the memory from the very first list item. As after the system startup only single pages are allocated, ony the release operation is an issue.

After the system startup only single pages are allocated. There is no need to avoid the fragmentation of the free memory. So the releasing operation could be simplified because the adjacent ranges need not to be merged. The sorting of the list is only needed to do the merging efficiently, so this also is not needed anymore. The released memory can be inserted at the head of the list, which only has a time complexity of $O(1)$.

Because the allocator can still cope with memory ranges that are larger than a single page and the initialization is also still done with a huge single memory block, the memory allocation at startup can be left unmodified. But, after longer operation, it is expected that the list will degenerate to a stack of free pages.

Part V.

Implementation

12. Kernel

This section discusses the main aspects of the implementation of the multiprocessor kernel. In 12.1 the message box system is described. The organisation of the TCB memory region is presented in 12.2. The implementation of the cross processor IPC is described in 12.3. Finally, the adaption of the kernel debugger to the multiprocessor system is presented in 12.4.

12.1. Message Box System

The message box system provides a communication and synchronization mechanism among the processors. It consists of an IPI handler, a matrix of message boxes, and functions to send requests and to explicitly handle requests while interrupts are disabled.

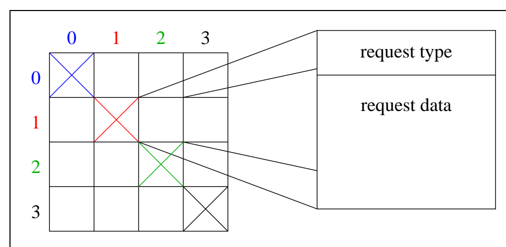


Figure 11: Message Box Matrix and Single Message Box

The shared data structure is an $n \times n$-matrix. It is allocated at startup and the pointer to it is stored in a way that all other processors can locate it. The matrix elements are message boxes and represent a communication channel between two processors. Message boxes allow to send a request in one direction and to return a reply to this request in the other. So the box for requests sent from processor A to processor B and the box for requests sent from processor B to processor A are different. Message boxes are implemented as variant records. The type field of the variant record marks whether the slot currently contains a request or a response. If the type field indicates a request the data field contains all parameters that are needed to handle the request. If the type field indicates a response the data field contains the return values from the request handler.

Sending a request begins with the preparation of the request parameters in the data field of the variant record. The second step is to set the request type field, so the destination processor can see that there is an incoming request in the message box. After this is done a notification is sent to the destination processor so the request handler can be invoked. The request handler can change the contents of the data field in order to put the return values there while handling the request. The type field is reset when finishing the request. The initiating processor waits until its request is finished.

However, while waiting it also serves incoming requests in order to avoid deadlocks that can occur if incoming requests are not handled while waiting for a response. In the situation of such deadlock a cyclic dependency of waiting for the own request to be handled would exist. Therefore the handling of a request must not depend on the completion of pending requests of other processors. Each processor serves

incoming requests while it waits for its own to complete. Cyclic dependencies cannot occur.

The cross processor IPC path requires that the awaited response is returned before any request that was sent later is handled. As the message boxes where incoming requests arrive are distinct from the message box where the awaited response arrives, it can happen that an incoming request is seen before the awaited response. To prevent this it is necessary to reorder the incoming requests with the awaited response. The ordering of the memory accesses to the message box type fields, which is a hardware issue, is the basis for the ordering of incoming requests and an awaited response. The arrival of the response is checked after the request was found. If the response already arrived, the handling of the request is postponed. If the response did not yet arrive, the request was definitely sent before it and can be handled immediately.

For the notification that a new request is in a message box IPIs are used. However, on IA32 processors IPIs get lost if the IPI waiting queue is full. Such a waiting queue consists of two waiting slots so only two IPIs can be pending. A third incoming IPI would be lost. On P6 architectures 16 IPI vectors share one waiting queue, on Xeon/P4 each IPI vector has its own waiting queue. So the IPI vector used for the message box system must be the only one that is used from a waiting queue. If there is only a single type of vectors in a waiting queue, it can be deemed the interrupts are aggregated instead of getting lost. If the interrupt gets aggregated, the request is already in the message box and an interrupt still is pending. So it is sure that the request will be handled.

12.2. TCB-Area

The TCB area is partitioned and reflects the partitioning of the task and the thread spaces. Every processor gets a single consecutive range of the TCB area and is responsible to manage its range.

On IA32 processors the boundary between the TCB subareas could be based on the granularity of page tables (4KiB) or of page directories (4MiB). For a more simple consistency management as described later 4MiB aligned boundaries are chosen. A memory zone of 4MiB can hold the TCBs for threads of 16 tasks.

The TCB area is the part of the virtual address space where all TCBs are located. The virtual address of a TCB of a certain thread can be computed from its thread id. During runtime the virtual memory is backed with physical memory on demand. A demand is recognized when some thread tries to access the TCB and generates a page fault because the TCB is not yet mapped. A zero page that resembles a TCB with the thread state *not existent* is mapped read-only if a read access caused the pagefault. As soon as a TCB is written to it must be backed with its own page. If during the handling of the write page fault it is found that the zero page already backs the TCB the existing translation is modifified. Due to the fact that on every processor another kernel address space is used, a consistency management between these address spaces is needed.

In *figure 12* the relations between the different page tables, page directories and their entries are shown. The sample entry belongs to the TCB of a thread running on *CPU 0*. Because every processor has its own kernel address space every processor has an own kernel page directory. When a processor populates a portion of its TCB range, it announces this new configuration via the shared *TCB master page directory*. Other processors update their page tables for this range from this sources when the triggered a pagefault after an attempt to access a TCB in that range. However, contrary to the managing processor the only have *read* permissions. Every processor manages exactly those entries that correspond with its local threads.

The consistency management uses two kinds of page tables: The first kind is a private page table. Private page tables are only used for the address translation for TCBs by the processor that owns these TCBs. If other processors also used this page table, a TLB-shootdown would be needed to announce the change of an existing mapping, especially from the *zero page* to another physical page. In comparision, shared page tables only allow changes from invalid translations to valid ones. These tables, though shared among the processors, are also managed by the processor responsible for the TCBs in the translated memory region. The processors use the shared page tables to translate addresses of non-local TCBs. To make these page

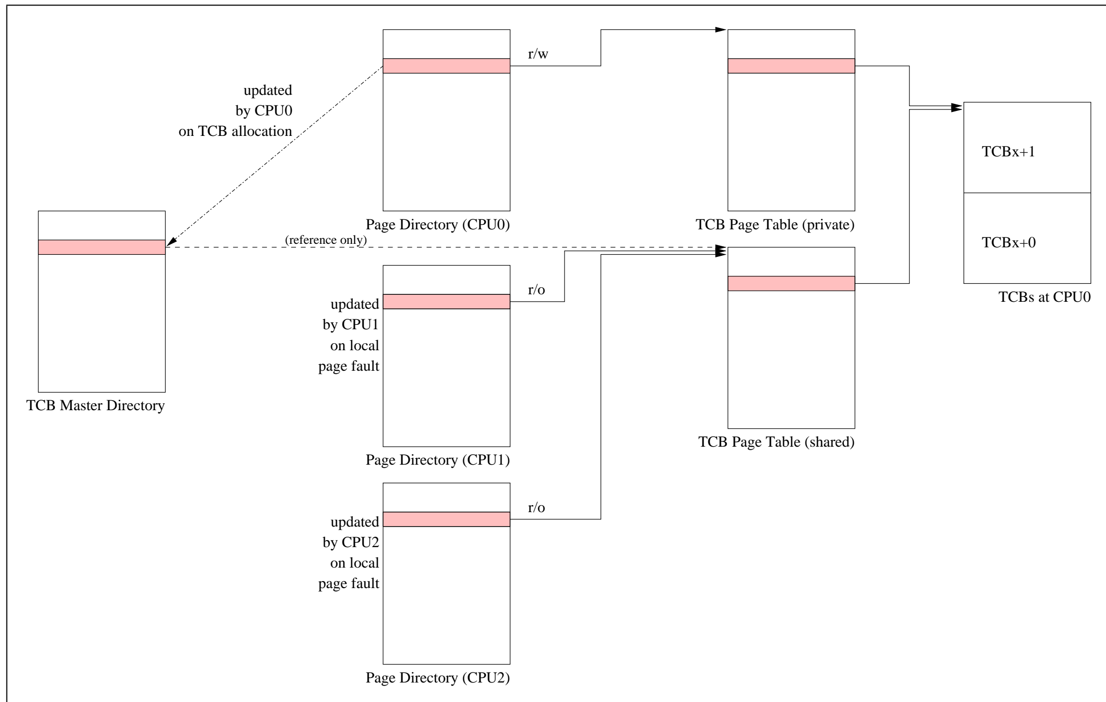


Figure 12: Mapping Structure with TCB Master Directory

tables public the managing processor registers them with the *TCB Master Directory*.

If a pagefault was triggered by an access to a local TCB, the first part of the handling is the same as in a single processor system. Only if the pagefault was a write fault, a translation for this TCB is created in the shared page table so other processors also have access to it. This could mean that a new shared page table has to be allocated and has to be registered with the *TCB Master Directory*.

If a pagefault was generated by an access to a remote TCB, the *TCB Master Directory* is checked for a shared page table that can be used for this translation. If there was no such page table, a pagefault resolve request is sent to the processor owning the TCB. If there was a shared page table for the faulting address but the translation itself was invalid, a pagefault resolve request is sent, too. A third possibility is that the translation is already present. This means that only the local page directory has to be updated.

While handling a page fault resolve request, it is possible that a private page table, a shared page table and a page for the TCB itself must be allocated. This is very undesirable during the request handling since the other processor is blocked during this time. So the requesting processor allocates up to three pages beforehand and passes the ownership, as described in 11.1.2, to the handling processor when sending the request. The handling processor passes the ownership of unused pages back to the requesting one. This may happen if there was another pagefault and some pages were already allocated. Further the requester can see whether the shared page table is already present, what implies that the private page table also is already present. If there is already a shared and a private page table the requestor need not to pre-allocate them.

12.3. Remote IPC-Path

This section describes the implementation of the cross processor IPC operation. In the first part (12.3.1) the different types of requests used for the message box system are described. The other part (12.3.2) describes

the atomic sequences that interact during a cross processor IPC operation.

12.3.1. Message Box Requests

There are three different request types needed for the cross processor IPC path. The *IPC Send Request* that start the IPC rendezvous, the *Wakeup Request* that wakes the sender and finishes the rendezvous when the receiver arrived after the sender, and the *Cancellation Request* that is used to cancel an already started rendezvous.

IPC Send Request

With the *IPC Send Request* the IPC sender requests the processor that hosts the receiver thread to deliver an IPC message to the destination thread. If this fails because the receiver is not yet ready to receive and the timeout is not zero then the sender thread is enqueued into the sender waiting queue of the destination thread. If the receiver was not ready and the timeout was zero, the message could not be delivered due to a timeout expiration. If the recevier was ready the message delivery succeeded. The request contains a pointer to the sender TCB, a pointer to the receiver TCB, and an indication whether the sender should be enqueued if the receiver is not ready to receive from it. The response contains the message delivery state that indicates whether the message was delivered successfully, the delivery is still pending, or the message delivery failed.

Wakeup Request

The *Wakeup Request* is sent from the processor of the receiving thread to the processor of the sending thread to indicate a change of the message delivery state. If the delivery state changed to delivered successfully the receiver got the message. But if the state was changed to delivery failed the operation was cancelled by the receiver. However, the latter cannot happen currently in Fiasco as the waiting sender list is not cleaned when a thread is killed. The request parameters are a pointer to the sender that should be waked and the new message delivery state. The response to this request does not contain any data.

Cancellation Request

Finally, the *Cancellation request* is initiated by the sender if the IPC operation is to be cancelled. The request handler tries to dequeue the sender from the sender waiting queue of the receiver. If the cancellation is too late the sender is already dequeued and therefore the cancellation will fail. The request contains a pointer to the TCB of the sender thread and a pointer to the TCB of the receiving thread as parameters. The return value is an information whether the dequeuing succeeded.

12.3.2. Atomic Sequences

Regular IPC Send Path

The cross processor IPC send path consists of three atomic sections.

The first atomic section of the regular IPC send path consists of three steps. The first step is the setup of the IPC operation. This includes to lookup the IPC partner and to test its existence, preparing the request in the message box, and preparing a potential receive operation. After this the second step is to send the request to the processor of the destination thread and to wait for the reply. The third step consists of evaluating

the response. If the response reports that the delivery failed the send path is left with a timeout error. If the message was successfully delivered the thread state is changed to receiving and the send path is left. If the message delivery is pending the thread is put to sleep. If the timeout is finite the timer is set up before going to sleep.

The second atomic section starts with the wakup of the thread. There are three reasons why the thread can be woken up: The thread was woken by the request handler because the delivery state changed to either failed or succeeded, a local `lthread_ex_regs` was executed on the thread, or the timeout expired. If the delivery state changed to delivered the send IPC path is left immediately in order to continue with the receive path or to return to the caller. If the delivery state changed to failed because the receiver thread was killed the IPC path is left and an according error code is returned to the user. In case the thread was woken by a timeout expiry or a local `lthread_ex_regs` the IPC operation must be cancelled. Hence a cancellation request is sent to the processor of the receiver.

The third atomic sections is executed when the response corresponding to the cancellation request was received. If the cancellation request was successfully handled the IPC path is left directly with the error code that represents the wakeup reason, i.e. IPC timeout or IPC cancel. But if the cancellation failed the thread must wait for the wakeup request sent from the other processor. It can only fail if the sender was already dequeued from the sender waiting queue. As dequeuing the sender and sending the wakeup request to the processor that hosts the sender are in the same atomic section the wakeup request is already visible in the incoming message box when the cancellation request fails. Therefore this request is handled directly. After this is done the path continues as if the request actually woke up the sender.

Request Handler at the Sender Side

Wakeup requests are the only request that have to be processed at the sender side. Before doing anything it is checked that the sender is polling for a receiver. If this is the case the message delivery state is set in the TCB so the sender can evaluate the wakeup reason. If the state signals a successful delivery and a receive operation should follow the send in the IPC system call then the thread state is changed to the receiving state. This must be done in this operation because this is the first point where the processor of the sender knows whether the send operation was successfully finished and it is the last point where it is synchronized with the receive operation on the side of the receiver. Further the timeout must be reset so the timeout cannot expire anymore. If the timeout hits after the change of the thread state, the receive operation could be cancelled due to the timeout of the previous send.

Timer Interrupt Handler at the Sender Side

This code sequence is already present in the single processor system. It is only mentioned to show the interaction with the cross processor IPC path and therefore only the relevant operation is described. If a timeout expires the thread that set the timeout is woken up and its state is reset so it can determine the reason for the wakeup.

`lthread_ex_regs` System Call at the Sender Side

This code is like the timer interrupt handler already present in the single processor system. During the system call the cancel flag is added to the thread state of the thread to which the system call was applied so the cancelled thread can determine the wakeup reason. After that this thread is woken up.

Regular IPC Receive Path

Whether the regular IPC receive path or the cross processor receive is taken is decided in the `ipc_receiver_ready` method. This method is called on waiting senders by a receiver that becomes ready to accept an IPC message. Because of this it can be asserted that the message was not yet delivered and the sender is sleeping. Further only if the sender was found in the waiting sender list this method is invoked and the sender is removed from the waiting queue. This part of the receive path will transfer the message as if the sender was a passive sender. It will peek the message from the TCB of the sender and store it to the TCB of the receiver. After delivering the message the sender wakeup request is sent to the processor of the sender.

Request Handler at the Receiver Side

One request type that is handled on the receiver side is the send request. If the receiver is already waiting for the sender, the message is just put into the receive buffer and the receiver is woken up. If the receiver is not ready to receive and the given timeout is not zero the sender is enqueued to the sender waiting queue of the receiver. If the timeout was zero, it just returns an delivery error.

The other request type that is handled at the receiving site is the cancellation request. The handler just tries to remove the sender from the sender waiting queue of the receiver and returns whether it succeeded.

12.4. Debugger Synchronization

For debugging purposes, especially for debugging this multiprocessor kernel implementation, it is very useful to run a kernel debugger along the kernel. This section is about the adaption of the kernel debugger to the multiprocessor environment. The main focus is only the adapt the debugger in order to facilitate the kernel development.

There are two basic ways to make the debugger working in the multiprocessor environment. The first is to run a debugger instance on the first processor and helper threads on the other processors. These helping threads gather the data required by the debugger and manipulate local items that are beyond the reach of the debugger instance on the first processor (e.g. debug registers). This would require the debugger modules to be aware of the multiprocessor environment.

A second way is to run a debugger instance on every processor. So the debugger instances run as independently as most kernel subsystems do. The implementation effort is restricted to the synchronization between the debugger instances.

The debugger synchronization must ensure that if the user is debugging the system excatly one debugger instance is driving the dialog between user and debugger. Of course, the user should be able to change the debugger instance that is driving the user dialog. A new debugger command is introduced for this purpose. Further the execution shall be interrupted on all processors when the debugger is entered.

The debugger synchronization model incorporates following states (see also *figure 13*):

normal operation This is the normal execution state of the system. No debugging is active.

handle int3 Local debugger requests are handled. This includes console output using the extended debugger interface and non-interactive control of debugger switches.

check spurious Handle spurious IPIs that occur after the debugger exit if IPIs from simultaneous entry requests are still pending.

entry barrier This state is to synchronize all processors. This state is left when all processors reached this state. This is needed to wait for processors that have interrupts currently disabled.

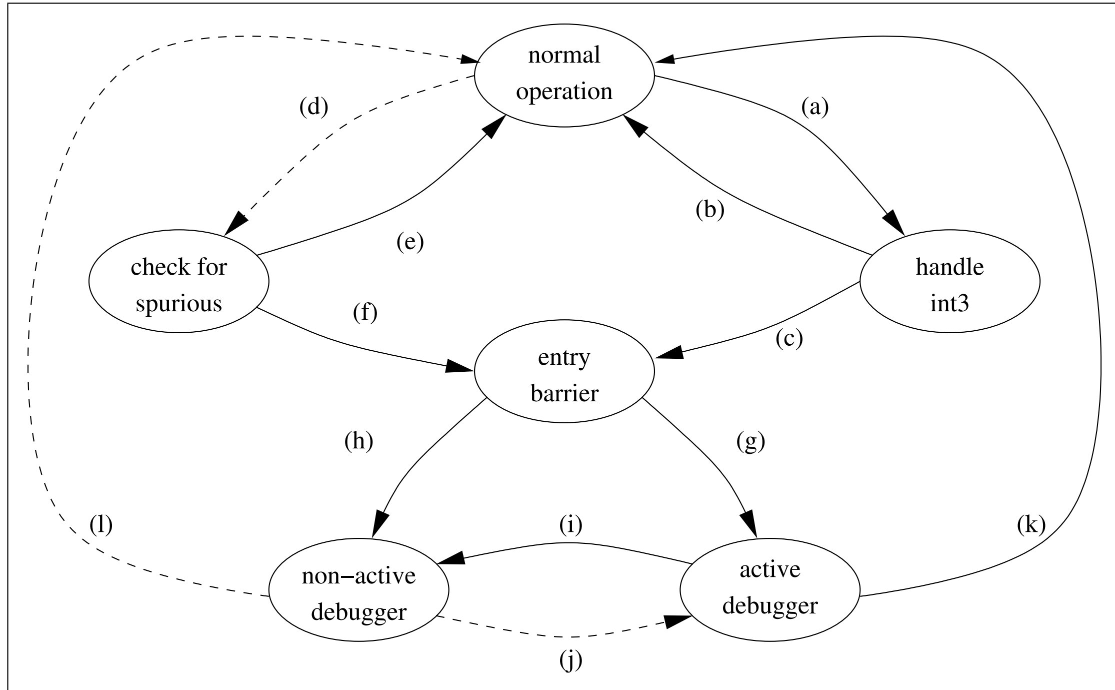


Figure 13: States of a Debugger Instance

active debugger In this state the debugger handles user input. At most one debugger instance can have this state at the same time.

non-active debugger The debugger instance waits for its activation or exit.

The transitions between the states (see also *figure 13*):

(a) **normal operation → handle int3**: This transition happens on `INT3`, an unhandled fault, or any other local event that leads to a debugger activation.

(b) **handle int3 → normal operation**: If the local debugger request could be handled already (e.g. by executing a non-interactive debugger command, by a (trace) buffer manipulation, or by outputting data to the kernel console) there is no need to enter the debugger anymore. So *normal operation* is resumed.

(c) **handle int3 → entry barrier**: If the local debugger request couldn't be handled there is still the need for a debugger entry. A flag is set indicating that a debugger entry is required by this instance. This must be done atomically as simultaneous requests can occur. The flags are stored in a flag field. These flags are needed later to determine whether any debugger instance and which debugger instances requested a debugger entry. After setting the flag an IPI is triggered to signal all other instances the entry request. After this the instance enters the *entry barrier* state.

(d) **normal operation → check for spurious**: This transition is asynchronous; it is triggered by an IPI designated for signalling debugger entry requests.

(e) **check for spurious → normal operation**: If the IPI was pending from the previous debugger entry the *normal operation* is resumed. This can happen on simultaneous entry requests as more than one IPIs are sent but only the first was handled. This situation can be detected by checking the entry request flags which are set before issuing the IPI in transition *(c)*.

(f) **check for spurious → entry barrier**: If the IPI was a regular debugger-IPI (at least for one processor the entry required flag is set) go to a wait state.

(g), (h) **entry barrier → active debugger, non-active debugger**: The *entry barrier* state can only be left when all processors reached it. The master processor (first system processor arbitrarily chosen as master) enters the *active debugger* state first. All other processors enter the *non-active debugger* state.

(i) **active debugger → non-active debugger**: This transition is triggered when the user inputs the debugger command for changing the active debugger instance. It will also trigger the reversed transition for the debugger instance which is chosen to be the new active instance.

(j) **non-active debugger → active debugger**: When the currently active debugger instance becomes inactive and this instance should be the next active instance this transition is taken.

(k) **active debugger → normal operation**: When the user issues the exit command of the debugger, the request flags are cleared and the normal operation is resumed.

(l) **non-active debugger → normal operation**: When the active debugger instance has cleared the request flags all inactive debugger instances resume *normal operation*, too.

13. Basic Services

This section describes the adaption of the basic services σ_0 and roottask. The adaption itself is, especially for roottask, preliminary. A multi-threaded design must be made for every L4env server and the usage of the roottask services must be examined to provide a final adaption. However, this is beyond the scope of this work. However, the basic services are adapted to start the demonstration and measurement application. This should give a reasonable base for further adaptions that are needed for the L4env servers.

13.1. σ_0

A σ_0 task is started on every processor with a memory mapping that is congruent. This leads to a programming paradigm of a multi-threaded server (see 8.1). For each thread a stack is necessary. These stacks are linked statically to simplify the startup process.

For internal memory allocation a simple startup allocator is used because only during the initialization memory is allocated by σ_0 itself. The initialization prepares an array of thread ids that is used to track the owner of a page. This array is dynamically allocated and registered as reserved memory. Afterwards it is initialized so that only non-reserved memory parts can be mapped to other tasks. As the mapping hierarchy is processor local this array must exist per processor. If every processor allocated its own management structure the startup allocator and the reserved memory registry had be changed in order to cope with multiple threads. For simplicity the first processor allocates all management data structures and registers them as reserved memory regions. After that it signals the other processors that the data structures are allocated. Now, every processor initializes its own management structures. For this only read access to the reserved memory registry is needed and therefore it need not be synchronized. Finally every σ_0 instance enters the server loop.

Processor local data structures are organized as an indexed array. The processors access these array with their own index. So all σ_0 threads are working independently from each other. There is no further synchronization needed.

13.2. Roottask

Roottask must be extended with additional functionality for the multiprocessor environment. The most basic functionality that is needed for the extension is the startup of tasks and the paging protocol on processors other than the first one.

Like the σ_0 server the roottask server is started on every processor. As σ_0 only can map normal memory idem-potently and the roottask server allocates all memory that is available from the σ_0 server, the virtual-to-physical translation of all roottask address spaces is identical. The roottask server uses the stub scheme that was descibed in section 8.1 as a possibility for multiprocessor aware servers.

The management data structures are only maintained on the first processor. During the initialization the roottask main thread and the stubs allocate all the memory they can get from their processor local σ_0 instance. The mappings will be exactly the same as the σ_0 instances can only map memory idem-potently and roottask gets all memory that is available. The last step of the initialization consists of starting the server thread on the first processor and the stub threads that handle cross processor task generation on the other processors. After passing a barrier that ensures that all roottask stubs got the memory from σ_0 the initialization thread enters the paging server loop.

The first stub thread is the paging protocol thread (or paging stub). It receives page fault IPC and explicit σ_0-protocol requests from tasks running at the processor. It then forwards the request to the real pager thread. The real pager thread handles the request as if it were local and returns it to the pager stub. Then the pager stub can simply reply to the request.

The second stub thread is the executer for task creations. If a task on the first processor requests roottask to create a new task that is located on another processor, the actual system call is not executed locally but a request to execute it is sent to the executer stub. Such requests do not fit in the two registers that are transfered by the cross processor IPC. Therefore the request data is stored in a data structure on the stack of the current thread and a pointer to this data structure is sent to the stub using cross processor IPC. When the IPC system call returns the stub has sent its reply to the request. Hence the stub does not use the request data structure anymore. As the stack is cleaned after the reply was evaluated it is a save storage location for the request.

Access rights are managed very simple in roottask. Either an object is allocated by a certain task and therefore it can access it, or the object is not allocated and the first requesting task gets it (i.e. the object is allocated to it afterwards), or an object is owned by another task and no access is allowed. Since there are multiple tasks that must have access to an object this scheme is insufficient. Since roottask mainly should start multi-threaded system services where all tasks belong to a single server and have the same access rights it would be sufficient to check the access rights as if the task of the server that runs on the first processor had sent the request. This means roottask must know about a mapping that provides a simple aliasing of tasks for the access right checking. For the time being this mapping simply aliases the n-th thread of a processor with the n-th thread of the first processor. Later this can be made dynamically by aliasing a new task with the creator of the thread.

14. Test and Debugging

14.1. Test Framework

For testing and debugging an in-kernel frame work was implemented in order to test low level functionality of the kernel. The kernel thread executes the test instead of starting σ_0 and roottask. The frame work provides a convenience interface to implement tests. This interface includes functions to start different test threads on the different processors, to startup further threads at the local processor, and to call system calls

from within the kernel. In order to be able to run different test setups without recompiling the kernel the setup to use is chosen with a kernel command line parameter.

14.2. Test Methods

In order to test a complex code path like the IPC path two approaches are used to trigger bugs and race conditions.

14.2.1. Modulated Delay Loop

When a race, which occur in a complex system setup, shall be detected, it may be difficult to trigger it in a test environment because of the code that is executed between the last synchronization point and the code where the race is located. This could mean that under test conditions one thread constantly arrives earlier than the other thread at the code parts that are critical. But in the complex system the executed code between the last synchronization and the operation that contains the race could be very long and therefore the drift between both threads is larger.

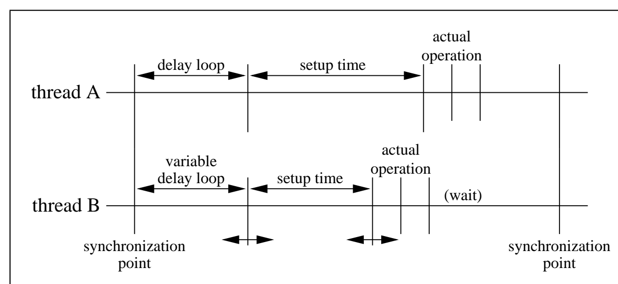


Figure 14: Scenario for Delay Loops to Detect Race Conditions

This time drift can be emulated under test conditions using modulated delay loops. *Figure 14* shows two debugging threads that perform a combination of operations that might contain a race condition. The time that is spent by the operations before they enter the critical part of the code is different. If the delay loops are not present (or need the same time) the critical code parts might not be executed concurrently. Therefore after the synchronization point both threads enter a delay loop. The one delay loop has a fixed time whereas the duration of the other delay loop can be modulated. During a test run the length of the modulated delay is swept from zero to a length about twice as long as the fixed delay. Therefore the starting points of the critical code path are shifted against each other. This method also compensates the inaccuracy of the synchronization operation that is used in the test environment.

14.2.2. Time Window

If there is the suspicion that a race happens because some other code executes between two instructions in the code path (e.g. due to preemption) it is possible to execute that code explicitly between those two instructions by using a kind of time window.

In *figure 15* thread A takes over the role of the thread containing the two instructions and thread B is the thread that executes the code that interferes between these two instructions. The synchronization can be done using a flag. When a thread has done its work it toggles the flag and waits until the other thread toggles the flag back. The flag is a memory location and the waiting could be done using spinloops. Of course, the tested code must be instrumented.

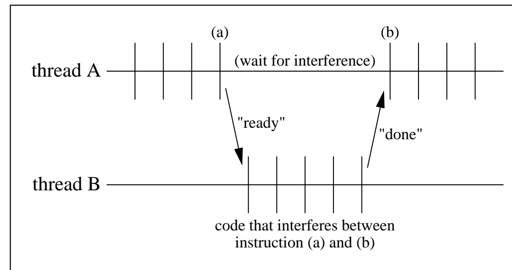


Figure 15: Scenario for Time Windows to Detect Race Conditions

14.3. Tests

This section describes the test that were implemented to test the new functionality of multiprocessor kernel and especially the cross processor IPC path.

14.3.1. Low Level Tests

Message Box System

This group tests the functionality of the message box system.

`test_msg_echo`: During this test the first processor requests the second to handle an echo request. The request handler at the second processor modifies the data in a way the first can check whether it was modified correctly (e.g. a bit-wise negation). This modification shall ensure that it can be verified whether the request was really handled at the second processor. After the modification the response is sent back to the first processor.

`test_msg_dbg`: This tests checkes if it is possible to enter the debugger while spinning in a waiting loop. This happens when the request handler generates a fault that leads to a debugger entry. For this a request is introduced that explicitly causes the debugger entry. The first processor sends the request to the second, which initiates the debugger entry. After leaving the debugger a response is sent back.

`test_msg_nested`: While spinning in the waiting loop requests from other processors must be handled. This test constructs a situation where a request must be handled. For doing this the request disregards the constraint not to send requests from within the request handler. Therefore it must check whether the processor already has an outgoing request pending. If there is no request pending a new request is sent to the processor that sent the first request and after receiving the reply a success information is returned in the response to the very first request. The first processor that drives the test can see whether the test condition occurred by checking the reply.

`test_msg_parallel`: This test is for the same purposes as the previous one. Each processor sends constantly a request to its successor in the enumeration. The last processor sends the request to the first one. The less processors are participating the more probable is a deadlock if the requests are not handled in the spinning loop. For two processors it was possible to trigger the deadlock if the request handling in the spinning loop was deactivated.

`test_msg_tcb_pf`: This test checks the correct handling of a pagefault happening during the access to a non-local TCB. The first processor simply reads TCBs of existing and non-existing threads that are not located at the first processor. After running the test the user must verify that the correct pages are mapped to the TCB area by using the kernel debugger.

Debugger Functionality

test_dbg_breakpoints: The test thread on the first processor reads from a certain virtual address and the test thread on the second processor writes to this virtual address. The more important things are the memory access itself. After starting the test the user must enter the debugger and set different breakpoints for the memory accesses. There are different combinations. One could, for example, set a breakpoint at the second processor that triggers if this address location is accessed (or written). This test was written in order to find a bug that was triggered when breakpoints were used for the first time during the development.

14.3.2. IPC-Path Tests

test_ipc_pingpong_call and **test_ipc_pingpong_nocall**: These two tests implement a simple pingpong scenario. The first processor sends an IPC message to the second processor and then the second processor sends another IPC message back. The first test uses the call and reply-wait versions of the IPC system call, the second test executes separate send and receive operations.

test_ipc_send: This test is intended to check whether the rendezvous is working. It uses modulated delay loops. The tested send and receive operations are blocking operations therefore the synchronization points are melted with the tested operations. This test triggered a multiprocessor bug in the IPC shortcut path.

test_ipc_noblock_send: This test uses the modulated delay loop method. The second processor consistently receives messages. The first processor tries to send an IPC message with timeout zero. This means the message must fail if the send operation is to early. For this reason the number of succeeded message transfers are counted at the sender and the receiver, if the result is the same the test was successful. If the send operation fails the receiver stays in the receive operation and therefore would never reach the next synchronization point. To prevent this the sender sends a special message that indicates that it was only sent to free the receiver so the receiver does not count it.

test_ipc_cancel: This test tries to find races between the cancellation of the send operation and the arriving receiver using a modulated delay loop. This means either the cancellation is early enough or the receiver gets the message. Like the previous test this test needs to count the successful IPC messages at sender and receiver side. The cancellation is done by a helper thread that runs on the sender processor.

test_ipc_timeout_send: This test is very similar to the previous test. This test tries to find races between the timeout expiry during the send operation and a receiver that becomes ready. The test also uses the method of the modulated delay loop but the fixed delay loop does not exist. It is replaced by the fixed timeout that is given for the send operation. With this test a bug was found in the original non-preemptible IPC path.

test_stress_cross_pingpong: This tests runs two pingpong tests at the same time using the first two processors. On every processor runs a ping thread that communicates with the pong thread that runs on the other processor.

test_stress_rcv_sync_{send,fail_send,mixed}: This is a group of tests that try to find bugs in the synchronization between senders that try to send to a single receiver. The send operation might be a blocking operation so it will succeed because the receiver gets ready anytime (`send`), or the send operation blocking time is limited by a timeout and the receiver will never actually receive a message (`fail_send`). The third form combines the two possibilities (`mixed`). This test mainly tests the queue operations of the waiting sender sender queue at the receiver and its surroundings. The stress tests found two bugs. The first was a wrong handling of timeouts in the original non-preemptible IPC path and the second was an incorrect implementation of the atomic transition from the cross processor IPC send path to the receive path within a single IPC system call.

14.3.3. Interference Tests

test_ipc_msg_*: This group of tests is constructed using the time window method. It is used to check whether certain requests are handled correctly while waiting for a pending IPC send request. In detail incoming requests for debugger entry (*_dbg), pagefault handling (*_pgfault), IPC send (*_send), IPC cancellation (*_cancel), and IPC wakeup sender (*_rcvd) are tested.

14.3.4. Performance Tests

There are also a couple of tests that measure the performance of low level operations. This includes hardware functionality like interrupts or cache times as well as functionality of kernel infrastructure like the message box system. The performance tests is described in more detail in the evaluation part (sections 15.2 and 15.1).

14.4. Debugging Illegal TCB Accesses

This debugging possibility was introduced to check whether the access constaints to TCBs of remote processors are followed after a bug in the IPC shortcut path was found. Therefore the shared page tables only contain references that make the page read only. In this way a processor can only read the remote TCBs. It is important that not for an instant the read only restriction is lifted. To enable the handling of the management of the sender waiting queues a bypass must be used. This bypass uses the fact that the pages for the TCBs are allocated using the low level memory allocator. The location of the page in the memory zone of this low level allocation is retrieved by a software page table walk and a simple computation. And the writable mapped page is accessed. This is a violation of the access rules to the low level memory allocation, but it is save because TCB mappings in the shared page tables are not altered after the first mapping.

14.5. Found Bugs

In this section the most time consuming bugs that where found during the implementation are described.

Incorrect Timeout Handling in The Original Non-Preemptible IPC Send Path

In the original non-preemptible IPC send path two bugs concerning the timeout handling in combination with preemption points where found. In both cases the timeout could expire during a preemption point.

In the first buggy code sequence the sender was first enqueued in the sender waiting queue at the receiver, then a preemption point followed, and after this the timeout was checked. If it expired, the code path is left due to the timeout. The problem is that during the preemption point the receiver can already find the sender in the sender waiting queue and reactivates it. For the receiver the rendezvous is over and it now waits for the sender to transfer the message. But the sender that was activated by the receiver encounters that the timeout already expired and leaves the IPC path with a timeout error. Thus the receiver will not get the message it waits for.

The second buggy code sequence was after leaving the polling loop of the sender where the message was successfully delivered. During a preemption point between this loop and the reset of, the timeout could expire and therefore reset the thread state. In consequence the kernel halted because of an assertion about the thread state. This bug was found by the test_ipc_timeout_send test because due to the sweeping delay the receive operation could be started delayed and therefore the timeout could be made to expire exactly during this preemption point.

Setup Check in The IPC Shortcut Path

Before actually executing the IPC shortcut path it must be checked whether all preconditions are fulfilled. One special precondition is that the TCB of the receiver thread must be already mapped. In the case that the TCB is already mapped the cost for this test should be negligible. For this a simple write access to the thread state of the partner thread is done. As the IPC path was originally preemptible and the thread state of the other thread must not be modified an atomic instruction is needed to read and write the thread state. An "`andl $0xffffffff, ...`" instruction was chosen. The operation itself will not change the content but a read and a write access are done. If a page fault occured due to this instruction (i.e. the TCB was not mapped) the page fault handler simply sets a flag and skips this instruction. And the test just evaluates the flag and knows whether the precondition is met. Unfortunately, this instruction is not atomic on multiprocessor systems. Further this instruction violates the constraint that it is not allowed to write to remote TCBs.

This combination leads to a lost update phenomenon if the two IPC operations are nearly simultaneously initiated. During the initialization of the receive the thread state will change back to an old state, at an arbitrarily point that occurs in the range of more than 40 lines of code. The most time for searching this bug was spent for the search of the error within these 40 lines that span across different file locations. After a bug in the receive path could be excluded and the debugger entry bug (described below) was corrected this bug was found very quickly.

A very simple correctio of this bug is to check first whether the destination thread is processor local, which is just another precondition for the shortcut path, and then wether the TCB to be mapped. The bug was first triggered by `test_ipc_send` because the modificated delay loop enabled to shift the two rendezvous points in a way so that these instructions were executed concurrently. To be able to track further access violations, i.e. write access to remote TCB, an extra infrastructure to detect illegal TCB accesses was established.

Erroneous Condition in The Debugger Entry Path

In order to debug the previously described bug there was the idea to track all accesses to the thread state. But when setting up breakpoints that shall trigger when the thread state is accessed the system hung. This was because the debugger entry path for the multiprocessor synchronization was faulty. As local request only the interrupt vector 3 was accepted. But the breakpoints use another interrupt vector and so no IPI was sent to signal the debugger entry. After assuming every interrupt that is forwarded to the debugger, except the interrupt denoting a remote entry request, is a local debugging request it worked.

Incorrect Transition Between IPC Cross Processor Send and IPC Receive Path

This bug goes down to a design blunder. In the first design and implementation it was assumed that the thread that is woken by a wakeup request is immediately scheduled but this is actually not ensured. So formerly the receive operation was setup after leaving the cross processor IPC send path. If the assumption that the thread is immediately scheduled were true, the wakeup and the code execution had been atomic.

The solution was to prepare the receive operation earlier and to change the thread state to receiving at that very moment the processor knows about the delivery even if it is in the request handler of the message box system.

This bug was triggered by the `test_stress_rcv_*` tests because these tests lead to situations where the sender thread is preempted by others and so cannot immediately go to the receive operation after being woken.

Part VI.

Evaluation

The test system contains two Xeon processor with HyperThreading support running at $3.4GHz$ and $4GB$ DDR-RAM. The system chipset is the Intel 82801EB/ER (ICH5/ICH5R).

The measurements are taken for cross processor operations that are done between two logical processors using the same processor core, labelled "Cross Sibling", and operations that are done between to processors that are really two different ones, labelled "Cross Package".

15. Low Level Functionality

In this section the performance of hardware functionality like cache line migration, low level synchronization between kernel threads, i.e. inter processor interrupts, memory flags, and the message box system is measured in order to get a clue about the expected performance of higher level operations. These tests are implemented using the in-kernel test frame work that was also used for debugging of low level functionality.

15.1. Cache Performance

There are five basic state transitions in the MESI cache coherence protocol. The tests are named after the prominent state changes of the transition.

e2s In the cache of processor A is an exclusive cache line and a processor B tries to read a memory location corresponding with this cache line. The cache line is subsequently shared.

e2m The cache of processor A holds an exclusive cache line and the processor B writes to a corresponding memory location. The cache line must be migrated to the processor B in order to modified it. After such transition the cache line is not present in the cache of processor A; and it can now be found in the cache of processor B having the state modified.

m2s This transition occurs if processor A has a modified cache line and another processor B whishes to read it. After the transition the cache line will be shared between both processors.

m2m This transition occurs if processor B wants to modify a cache line after processor A has modified it. The cache line is only present in the cache of processor B and is in the modified state, afterwards.

s2m If two or more processors share a cache line and a processor B wants to modify it, the cache line will be invalidated on all other caches. This, as all modifications to cache lines, leads to the situation that the cache line is a modified cache line in the cache of the modifying processor.

The transitions from one protocol state to another not only consists of the synchronization between the processors, they also contain possible memory write backs, reads, bus snooping operations, or some combinations of it. The actual execution times depends largely upon the specific MESI implementation that can be, and most probably is, optimized (e.g. extended with further sub-states of the four main states). Further, the Xeon processor also contains optimizations that recognize certain access patterns and prefetch data that is probably needed next.

The measurement results are shown in *table 1*.

Test Name	Cross Sibling	Cross Package
e2s	78	77
e2m	126	129
m2s	7	77
m2m	13	136
s2m	13	136

Table 1: Cycles Needed For Cache Protocol State Transitions

15.2. Cross Processor Synchronization

Five basic cross processor synchronization types are evalutated. The name of the tests is derived from the basic functionality that they test.

ii In this test the rount trip time of the IPI synchronization is measured. Therefore processor A sends an IPI to processor B. The interrupt handler on processor B sends back an IPI to processor A. The interrupt handler on processor A writes the current time stamp into a variable where the test thread can read it from.

if Processor A resets a synchronization flag, sends an IPI to processor B, and waits for the flag to be set. The interrupt handler of processor B solely acknowledges the interrupt and sets this synchronization flag.

ff1 Processor A resets a synchronization flag and waits for the flag to be set again. Processor B is the antagonist that waits for the flag to be reset and sets it again. For this synchronization a single flag is used.

ff2 For both processors A and B a synchronization flag is used. Every processor only writes to the own flag. Processor A toggles its own synchronization flag and waits for the remote flag to change. Processor B does exactly the same.

msg This test measures the time that is needed for a dummy message in the message box system. The message box system provides a no-op request type where the response is sent back immediately.

Test Name	Cross Sibling	Cross Package
ii	5720	5245
if	2771	2989
ff1	472	584
ff2	369	938
msg	3474	3791

Table 2: Cycles Needed For Cross Processor Synchronization

The measurement results are shown in *table 2*. In the following paragraphs some results are discussed.

The IPI round trip time was measured with the `ii`-test. For processors with HyperThreading support there are two separate local APIC (see [15] section *7.8 Intel Hyper-Threading Technology Architecture*). To send the IPI between two logical processors that use the same processor core the IPI needs to be processed by the APICs in their own processing speed, even if the IPI-message is not seen on the bus. Therefore it was expected that the round trip time between logical processors that use the same core and the round trip time between physically distinct processors are not significantly different. That the HyperThreading IPI is slower than the cross processor IPI could be explained by the HyperThreading itself as the executing of the logical processors is mutually hampered.

For subsequent accesses to the same flag by multiple processors there seems to be an extra performance penalty because the cross processor synchronization does need more time than it would be justified by the measured cache times. As already mentioned the measured cache times are expected to be lower than the overall times needed for an operation that is executed on a longer code path and not in a tight loop. So the flag synchronization gives a more realistic estimation for the cache transfer times.

16. IPC Performance

16.1. Cross Processor IPC

This test measures the round trip time of the cross processor IPC operation. The test consists of two parts. Between these parts the workload in the system is varied. In the first part no additional workload is running. It will measure the time needed for the cross processor IPC itself. In the second part of the test an intra-task short IPC pingpong is running on one processor that is used for the measurement. Besides the time for the cross processor IPC it will also include the time for preemption of the processor local pingpong by the measured one.

Expected Measurement Values

Sending a cross processor IPC message consists of checking whether the IPC partner exists, sending a request to the other processor, transfering the actual IPC message, and operations that also have to be done in the processor local IPC like system entry and exit, or dequeuing and enqueuing at the run queue.

When checking for the existence of the IPC partner a cache line is migrated, i.e. an **m2s** transition takes place. Further an **s2m** transition is expected when the IPC partner changes its thread state again.

The no-op message did not contain any payload. Therefore additional costs for the cache line migration of the payload is expected. The following cache line migrations are expected: **s2m** when the payload is written by the initiator, **m2s** when it is read by the handler, **s2m** when the handler composes its response, and finally **m2s** when the initiator evaluates the response, which leaves the caches in the state that was assumed for the firt step.

For the message transfer further cache line migrations are expected: The register-only short IPC fits into a single cache line. This cache line is migrated to the processor of the receiver and thus has a **m2s** transition. As the buffer for sent messages and received ones is the same this buffer was formerly read in the progress of the pingpong-test and thus yet another **s2m** transition takes place.

Test Name	Cross Sibling	Cross Package
8×**s2m**	56	616
8×**m2s**	104	1088
2×**msg**	6948	6982
total	7108	8686

Table 3: Expected Round Trip Times for Cross Processor IPC

For the round trip time the operations above are done twice. The estimation of execution times is shown in *table 2*. However, time that is needed for the execution of the code path is not included.

Test Results

Table 4 shows the meassured arithmetic mean for the different tests. What can be seen clearly is that

Test Name	Cross Sibling	Cross Package
no additional load	15258	31764
pingpong at one CPU	15280	44665

Table 4: Measured Round Trip Times for Cross Processor IPC

the estimation is far away from the measured values. The most operations need around the double time of the estimated one. A detailed performance analysis will be needed to discovers the bottlenecks in order to optimize the performance.

16.2. Influence of Workload on Local IPC

In this test the round trip time of an intra-task short-IPC is measured. The test is run with different workload variants: (a) No other workload is started, (b) only a processor local workload is started on a processor other than the measurement processor, and (c) a cross processor pingpong is started that involves the measurement processor. The cross processor pingpong thread runs at the same priority as the measurement thread.

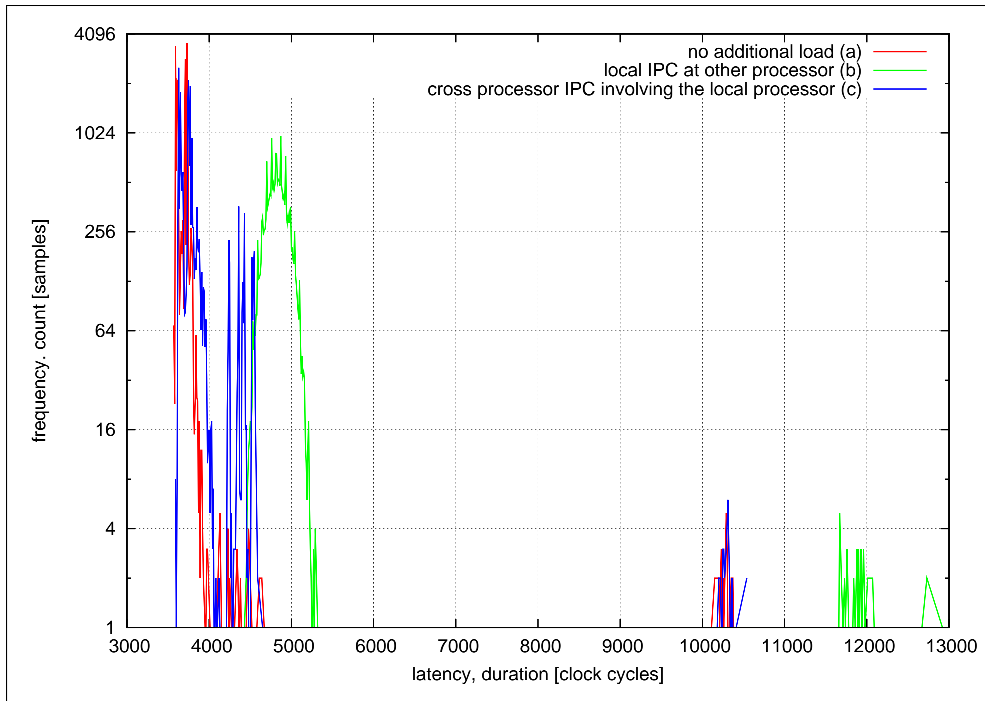


Figure 16: Influence of Different Workload on Local IPC (Cross Sibling)

Figure 16 shows the results when using siblings with the same processor core for this test. It can be seen in the curve of test (b) how the siblings hinder each other in the execution. The peaks in the curve of test (c) between 4000 and 4500 cycles represent samples where the handling of the *IPC send request* interrupted the execution of the processor local IPC. Peaks that are above 10000 cycles are probably the influence of the timer interrupt.

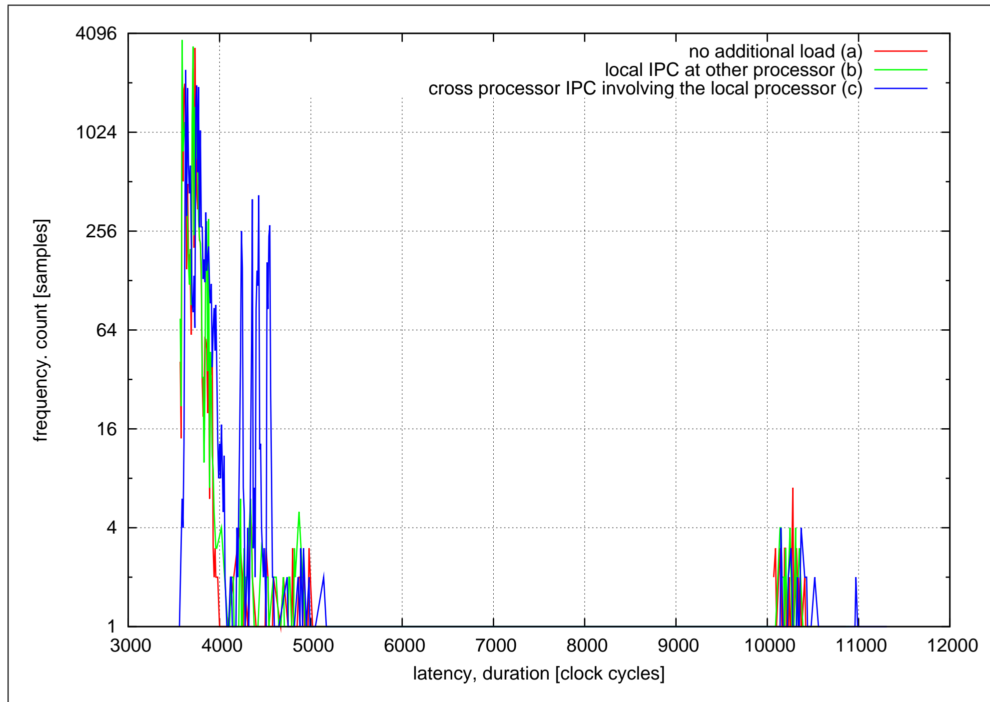


Figure 17: Influence of Different Workload on Local IPC (Cross Package)

Figure 17 shows the results when processors from different packages were used. It only differs for the test (b) because the used processors does not hinder each other in this case.

16.3. Local IPC on MP-Kernel vs. IPC on UP-Kernel

To determine the performance impact to the processor local operation that occurs due to the multiplexing of local and cross processor operations behind the same interface, the standard measurement programm `pingpong` is used. The measurements on the unmodified uniprocessor kernel and on the multiprocessor kernel are compared.

Test Name	Original Fiasco		Fiasco-MP	
	shortcut	no shortcut	shortcut	no shortcut
`int30/warm`	2514	2904	2492	2694
`sysenter/warm`	1331	1663	1298	1590

Table 5: Influence of Modification of Processor Local IPC Path

As it can be seen in *table 5*, the modified IPC path is even faster. The reason could be some anomaly in the pipeline, cache, or brach prediction.

17. Interrupt Latency

In order to measure the interrupt latency at least two processors are needed. One processor generates an interrupt event by sending an IPI and measures the time needed for the reaction. The other processors

handles the interrupt and reacts by setting a memory flag.

For this test different workloads are executed on the interrupt handling processor: (a) no work load at all, to have an reference basis when evaluating the influences; (b) continous allocation and deallocation of pages with the low level memory allocator; (c) a processor local pingpong; and finally (d) a cross processor pingpong with a third processor. This workload is running at full speed.

But between the single measurements a certain time is waited. This time is varied to prevent an interference between the rates of the workload and the measurement.

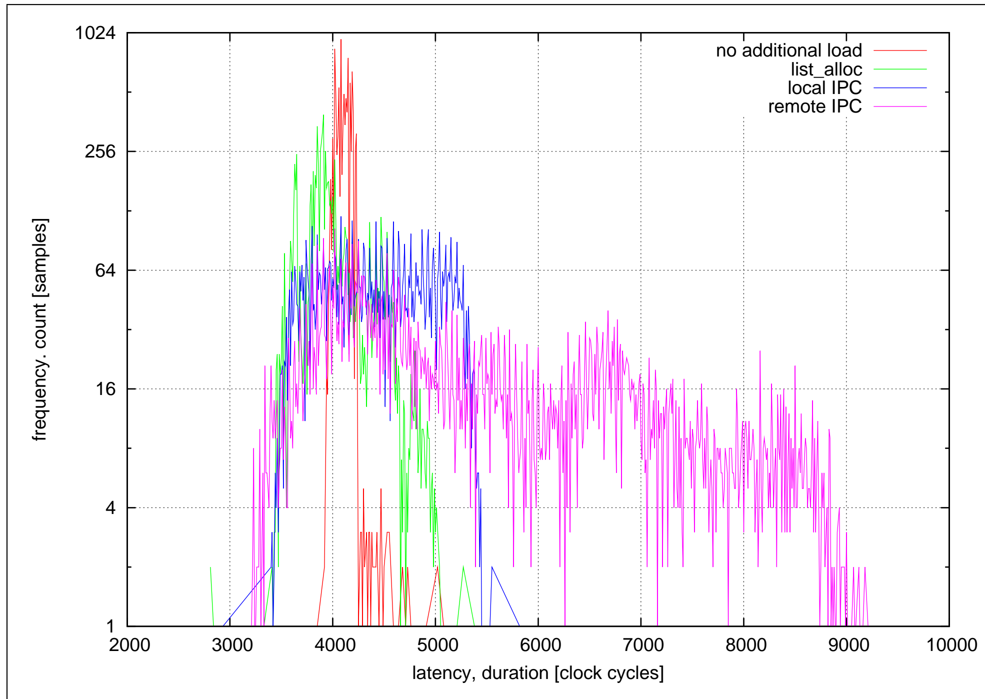


Figure 18: Interrupt Latencies at Different Workloads

In *figure 18* the results are shown for the test where the measuring and the interrupt handling processor share no processor core. It can be seen that the modified list allocator does not influence the interrupt latency as much as the non-preemptible part of the local IPC path.

As it was expected, cross processor IPC increases the interrupt latency very much. This is for the interrupt disabling while waiting for the handling of a request. However, this might be reduced a little bit by introducing preemption points into the cross processor IPC path.

Part VII.

Conclusion and Future Work

This work showed that it is possible to instantiate the subsystems that exist in the current Fiasco kernel per processor and to run them independently. The proposed model was sufficient to implement the demonstration and measurement application. There is also a reasoning that it is sufficient for L^4Linux. But this is still to be proofed practically.

Some extensions of roottask for the multiprocessor environment are better designed and implemented with the extensions of DICE, the IDL compiler for L4Env, because it is used for the roottask server thread and now must be used for the multiple roottask server threads.

As the performance did not meet the expectations it must be investigated what are the reasons and where the implementation can be optimized or even which parts have to be redesigned.

During the work the Fiasco main branch had many changes, including major ones. As the implementation is relativly unintrusive it could be a good way to make a forward-port of the alterations. For this following problems still must be solved:

The first thing that is incompatible is the new handling of 4GiB and especially the implications for the KIP handling. Formerly memory that was allocated using the low level allocator was visible to σ_0. This assuption is broken as this memory is now located beyond 3GiB. The KIPs for the processors are allocated this way and therefore they are not accessible to σ_0.

Another problem that is not solved is that the 60MiB that Fiasco reserves for kernel memory is not sufficient if the main memory is bigger. A solution could be to use a memory allocate that works for physical memory that is not visible in the kernel address space

Part VIII.

Summary

In this work a multiprocessor model was evolved that explicitly exposes the multiprocessor view to applications. Further the plausibility of this model was checked by modelling the intended workload on top of it. The user model was then implemented in the Fiasco microkernel and the basic services σ_0 and roottask were adapted.

Among others, the implementation in Fiasco consisted of a new cross processor IPC path. This new IPC path has limited functionality compared to the original Fiasco IPC path. A main issue was the nearly seamless integration of the new cross processor IPC path into the existing. This especially means the cooperation of the cross processor send with the open receive that can receive from the local processor as well as from other processors. Another point of this integration was the atomic transition from the cross processor send operation to the receive operation in the IPC system call.

During the debugging stage a couple of bugs were found in the new cross processor IPC path and another two were found in the already existing, and now processor local IPC path.

The much time was spent for implementation and especially for debugging. The debugging was necessary because cross processor IPC path was completely new and therefore it needed to be tested thoroughly.

Part IX.

Appendix

A. Glossary

CPU
: central processing unit: Commonly used as synonym for processor.

IPC
: inter process communication: Inspite of its name threads are communicating in L4 not processes, which do not even exist in L4.

IPI
: inter processor interrupt: A hardware signalling mechanism in multiprocessor system to influence the execution flow of another processor.

MESI
: modified, exclusive, shared, invalid: The four states that a cache line when using the *MESI*-protocol. Cache coherence protocols are commonly named after the possible states of the cache lines.

RCU
: read-copy-update: A synchronization method that favors read accesses. A write operation is finished when all readers see the modification, which might take some time.

TCB
: *here:* thread control block: A datastructure that holds metadata about a thread. Metadata can be the current thread state, the execution state, queueing information, etc.

TLB
: translation look-aside buffer: A cache for virtual memory address to physical memory address translations. Every processor has its own TLB and hardware do not implement any coherence mechanism, neither between processors nor between the TLB and the pagetables.

B. Bibliography

References

[1] L4env - Documentation. Available from URL: http://os.inf.tu-dresden.de/l4env/docu.xml. 8

[2] L4Ka::Hazelnut. Available from URL: http://www.l4ka.org/projects/hazelnut/. 1

[3] L4Ka::Pistachio. Available from URL: http://www.l4ka.org/projects/pistachio/. 1

[4] NICTA::Pistachio-embedded. Available from URL: http://ertos.nicta.com.au/research/l4/embedded.pml. 1

[5] Marcus Völp Bernard Kauer. *L4.sec Preliminary Kernel Reference Manual*. TU Dresden, October 2005. 1

[6] U. Dannowski, J. LeVasseur, E. Skoglund, and V. Uhlig. L4 experimental kernel reference manual, version x.2. Technical report, 2004. Latest version available from: http://l4hq.org/docs/manuals/. 1

[7] Andreas Häberlen. Managing Kernel Memory Resources from User Level, April 25 2003. 1

[8] M. Hohmuth. Linux-Emulation auf einem Mikrokern. Master's thesis, TU Dresden, August 1996. In German; with English slides. Available from URL: http://os.inf.tu-dresden.de/~hohmuth/prj/linux-on-l4/. I, 8

[9] Michael Hohmuth. The Fiasco kernel: Requirements definition. Technical Report TUD–FI–12, TU Dresden, December 1998. Available from URL: http://os.inf.tu-dresden.de/papers_ps/fiasco-spec.ps.gz. 2

[10] Michael Hohmuth. The Fiasco kernel: System architecture. Technical Report TUD-FI02-06-Juli-2002, TU Dresden, 2002. I, 2

[11] Michael Hohmuth. *Pragmatic nonblocking synchronization for real-time systems*. PhD thesis, TU Dresden, Fakultät Informatik, September 2002. 11.2.1

[12] Michael Hohmuth and Michael Peter. Helping in a multiprocessor environment. In *Proceedings of the Second Workshop on Common Microkernel System Platforms*, 2001. Available from URL: http://os.inf.tu-dresden.de/~hohmuth/prj/smp-helping.pdf. 11.2.1

[13] Intel Corp. *IA-32 Intel Architecture Opimization Reference Manual*, June 2005.

[14] Intel Corp. *Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture*, September 2005.

[15] Intel Corp. *Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide*, September 2005. 15.2

[16] Intel Corp. *Intel Architecture Software Developer's Manual, Volume 2A: Instruction Set Reference A-M*, January 2006.

[17] Intel Corp. *Intel Architecture Software Developer's Manual, Volume 2B: Instruction Set Reference N-Z*, January 2006.

[18] Bernhard Kauer. L4.sec Implementation - Kernel Memory Managment. Master's thesis, TU Dresden, May 2005. 1

[19] Adam Lackorzynski. L^4linux on l4env. Großer Beleg, TU Dresden, December 2002. I, 8

[20] Adam Lackorzynski. L^4linux porting optimizations, March 2004. I, 8

[21] 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. 1

[22] 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. 1

[23] 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. 1

[24] J. Liedtke. L4 nucleus version x reference manual (x86). Technical report, September 1999. 1

[25] Michael Peter. Portierung des fiasco μ-kernel auf smp-systeme. Großer Beleg, TU Dresden. 5, 7.2

[26] René Reusner. Implementierung eines echtzeit-ipc-pfades mit unterbrechungspunkten für l4/fiasco, July 2005. 11.2.1

[27] Udo Steinberg. Quality-Assuring Scheduling in the Fiasco Microkernel. Master's thesis, TU Dresden, March 2004. 2

[28] Volkmar Uhlig. *Scalability of Microkernel-Based Systems*. PhD thesis, Universität Karlsruhe (TH), Fakultät für Informatik, June 2005. 4, 7.2

[29] Marcus Völp. Prototypical design and implementation of l4-smp microkernel mechanisms. Technical report, TU Karlsruhe, April 2002. 3, 7.1.1, 7.2