



Universität Karlsruhe (TH)  
Institut für  
Betriebs- und Dialogsysteme  
Professor Dr. Alfred Schmidt

# Design and implementation of the Recursive Virtual Address Space Model for Small Scale Multiprocessor Systems

Marcus Völp

Diplomarbeit

Verantwortlicher Betreuer: Dr. Kevin Ephinstone  
Betreuender Mitarbeiter: Volkmar Uhlig

September 2002



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

I declare to have written this thesis independently and that no unmentioned literature has been used.

Karlsruhe, den 11. September 2002

---

Marcus Völp



# Contents

<b>1</b>	<b>Introduction</b>	<b>17</b>
1.1	Proposed Solutions . . . . .	18
1.2	Outline . . . . .	18
<b>2</b>	<b>Background</b>	<b>19</b>
2.1	Mechanisms for Page Based Virtual Memory . . . . .	19
2.1.1	Terminology . . . . .	19
2.1.2	Linear Page-Tables . . . . .	20
2.1.3	Multi-Level Page-Tables . . . . .	20
2.1.4	Guarded Page Tables . . . . .	21
2.1.5	Inverted Page-Tables . . . . .	23
2.2	Translation Lookaside Buffer . . . . .	23
2.3	Small Scale Multiprocessors . . . . .	24
2.4	Recursive Virtual Address Space Model . . . . .	26
2.4.1	Map . . . . .	26
2.4.2	Grant . . . . .	27
2.4.3	Unmap . . . . .	27
2.4.4	Overmap and Fast Overmap . . . . .	29
2.4.5	Page Reference Information Retrieval . . . . .	30
2.4.6	Multiple Hardware Page-Size Support . . . . .	31
2.4.7	Fundamental Consistency Requirements . . . . .	33
2.5	Mapping Database . . . . .	34
2.5.1	Structure of the Mapping Database . . . . .	35
2.5.2	Requirements and Constraints . . . . .	36
2.6	Unbounded Priority Inversion . . . . .	39
2.6.1	Definition . . . . .	39
2.6.2	Unbounded Priority Inversion in Address Space Construction . . . . .	39

2.6.3	Unbounded Priority Inversion of Preorder Unmap . . . . .	40
2.6.4	Generalization . . . . .	42
2.6.5	Helping . . . . .	43
2.7	Summary . . . . .	45
<b>3</b>	<b>Fundamentals and Related work</b>	<b>47</b>
3.1	Existing Implementations of the Recursive Virtual Address Space Model	47
3.1.1	The $\mu$ -kernel Approach . . . . .	47
3.1.2	L4 $\mu$ -kernel . . . . .	48
3.1.3	Implementations of the Recursive Virtual Address Space Model .	48
3.2	Synchronization . . . . .	49
3.2.1	Overview of Synchronization Techniques . . . . .	50
3.2.2	Locking policies . . . . .	51
3.2.3	Uniprocessor Synchronization . . . . .	51
3.2.4	Scheduler-Conscious Synchronization . . . . .	52
3.3	Summary . . . . .	52
<b>4</b>	<b>Mapping-Database Design</b>	<b>53</b>
4.1	Mapping-Tree Representation . . . . .	54
4.1.1	Mapping Node . . . . .	54
4.1.2	Requirements . . . . .	54
4.1.3	Representations . . . . .	58
4.1.4	Summary Tree Representation . . . . .	70
4.2	Concurrency and Consistency . . . . .	71
4.2.1	Granularity . . . . .	72
4.2.2	Synchronization Techniques . . . . .	74
4.2.3	PTE Lock . . . . .	77
4.3	Preemptability and Progress . . . . .	78
4.3.1	Restart of Unmap: Problem Analysis . . . . .	78
4.3.2	Preempted-Thread List . . . . .	79
4.3.3	Token-Based Preempted-Thread List . . . . .	81
4.3.4	Summary . . . . .	83
4.4	Multiple Page-Size Support . . . . .	85
4.4.1	Split Mapping . . . . .	85
4.4.2	Partial Unmap . . . . .	85
4.4.3	Root Array Structure . . . . .	87
4.4.4	Frame Locks Revisited . . . . .	88

4.4.5	Restart-Point Tracking . . . . .	92
4.4.6	Space Requirements . . . . .	93
4.5	Summary of the Design . . . . .	94
<b>5</b>	<b>Experimental Results and Analysis</b>	<b>97</b>
5.1	Experimental Setup . . . . .	97
5.2	Performance of the Address Space Modifiers . . . . .	98
5.2.1	Map Performance Comparisson . . . . .	98
5.2.2	Unmap Performance Comparisson . . . . .	100
5.3	Interrupt Latency . . . . .	102
5.4	Analysis . . . . .	103
<b>6</b>	<b>Conclusion and Future work</b>	<b>105</b>
6.1	Conclusion . . . . .	105
6.1.1	Approach . . . . .	106
6.1.2	Summary of Results and Conclusions . . . . .	106
6.2	Future Work . . . . .	108
<b>A</b>	<b>Data Structures</b>	<b>111</b>
A.1	Mapnode . . . . .	111
A.2	Preemption token . . . . .	114
A.3	Root overrun token . . . . .	114
A.4	Multiple pagesize support . . . . .	115
A.4.1	Dual node . . . . .	115
A.4.2	Rootnode . . . . .	115
A.4.3	Mid-Rootnode . . . . .	116



# List of Figures

2.1	Linear Page-Table . . . . .	20
2.2	Multi-Level Page-Table . . . . .	21
2.3	Guarded page-table . . . . .	22
2.4	<i>Inverted Page-Table</i> . . . . .	24
2.5	<i>Symmetric Multi-Processor Hardware</i> . . . . .	25
2.6	Mapping of a Page . . . . .	27
2.7	Granting of a Page . . . . .	27
2.8	Unmapping a Page . . . . .	28
2.9	Overmapping a Page . . . . .	29
2.10	Split Mapping . . . . .	32
2.11	United Mapping . . . . .	32
2.12	Partial Unmap . . . . .	33
2.13	Pinning . . . . .	34
2.14	Modification of Map on the Mapping Database . . . . .	35
2.15	Modification of Grant on the Mapping Database . . . . .	36
2.16	Modification of Unmap on the Mapping Database . . . . .	36
2.17	Unbound priority inversion of Concurrent Pre-order Unmaps . . . . .	41
2.18	Unbounded Priority Inversion in other Traversal Methods . . . . .	42
3.1	Left child - Both siblings . . . . .	49
4.1	Page Reference Information . . . . .	56
4.2	Mapping-Database Representation: LL . . . . .	58
4.3	<i>Pointers of LL</i> . . . . .	59
4.4	Mapping-Node Insertion in LL . . . . .	60
4.5	Mapping-Node Deletion in LL . . . . .	61
4.6	Mapping-Node Reallocation in LL . . . . .	61
4.7	Tree-Oriented Representation . . . . .	63

4.8	Operation-Oriented Representation . . . . .	63
4.9	Mapping-Database-Representation: LL-Tree . . . . .	64
4.10	<i>Pointers of LL-Tree</i> . . . . .	65
4.11	Child-Link Update in LL-Tree . . . . .	66
4.12	Mapping-Node Reallocation in LL-Tree: Child-Link Update . . . . .	66
4.13	Mapping-Node Reallocation in LL-Tree: Sibling-Link Update . . . . .	67
4.14	First-Child-List shares Left-Most-Leaf . . . . .	68
4.15	Mapping-Database Representation: LL-O1 . . . . .	69
4.16	<i>Pointers of LL-O1</i> . . . . .	70
4.17	Anomaly in LL-O1 Representation . . . . .	70
4.18	Mapping-Node Reallocation in LL-O1 . . . . .	71
4.19	Two Phase Locking Protocol for LL . . . . .	73
4.20	Page-Table-Entry Lock . . . . .	77
4.21	Restart-Point of Preempted Unmap . . . . .	79
4.22	Preempted Thread List . . . . .	79
4.23	Starvation in Thread List . . . . .	80
4.24	Token-Based Preempted-Thread List . . . . .	81
4.25	Partial Unmap with Page Split-Up . . . . .	86
4.26	Separation of Multiple Page-Sizes with Root Arrays . . . . .	88
4.27	Tree Representations for Multiple Page-Sizes: LL . . . . .	89
4.28	Tree Representations for Multiple Page-Sizes: LL-Tree . . . . .	90
4.29	Tree Representations for Multiple Page-Sizes: LL-O1 . . . . .	91
4.30	Traversal of Multi Page-Size Mapping-Database . . . . .	92
5.1	Performance of the Map Operation . . . . .	99
5.2	Performance of Wide Unmap . . . . .	100
5.3	Performance of Deep Unmap . . . . .	101
5.4	Preemption Points of Unmap in the Interrupt-Latency Experiment . . .	102
5.5	Interrupt Latency . . . . .	103

# List of Tables

4.1	Content of Mapping Node . . . . .	54
4.2	Summary of Representations . . . . .	71
4.3	Payload Information of a Mapping Nodes . . . . .	72
4.4	Space Requirements for Mapping Nodes . . . . .	94

## Abstract

Standard virtual memory management (VMM) and protection policies are often not suitable to serve the requirements of special applications such as database management and multi-media. Two approaches have been proposed to support the differing demands of those kind of applications:

- *extension* of the standard policies with application specific policies, and
- *externalization* of virtual memory management and protection policies to application specific managers.

An *externalization* of VMM policies to user level managers makes it possible to select those policies that best serve those requirements. In addition to that, *externalization* reduces the complexity of inside kernel memory management. The *Recursive Virtual Address Space Model* is such an approach of *externalization*.

Undesired behavior and unnecessary complex or slow behavior of existing implementations of this model, however, limit its applicability. Such behavior includes:

- *long interrupt latencies*, and
- *unbounded priority inversion*.

The first is caused by non-preemptively executing those operations modifying address spaces as it is done in some existing implementations. Other implementations allow for those operations to be preempted. This avoids long interrupt latencies, but those implementations are prone to unbounded priority inversion. To avoid this unbounded priority inversion, those implementations apply complex timeslice donation and “*helping-schemes*” [HH01]. In Section 2.6.5, we will point out that those “*helping-schemes*” imply system behavior that cannot be tolerated in all systems. In particular in small scale multiprocessor systems, this may lead to a behavior that is not desirable in all systems. A frequent migration of threads that get “helped” is an example of such a behavior. We aim to avoid “helping” in our solution.

This thesis investigates the problem of unbounded priority inversion in preemptable implementations of the *Recursive Virtual Address Space Model*. We propose new methods to control concurrent address space construction, that

- *do not lead to long interrupt latencies*,
- *do not lead to unbounded priority inversion*,
- *do not lead to starvation*, and
- *do not require helping*.

In particular, we propose:

- *preemptable post-order traversal*,
- *roll forward combined with scheduler-conscious locking*, and
- *efficient restart-point tracking*

to achieve those objectives.

In this thesis we describe and evaluate those techniques in the implementation of the *Recursive Virtual Address Space Model*. We show that a comparable performance to non-preemptive implementations can be achieved for address space construction. The costs of revoking memory from address spaces, however, show an increase by 12.2% compared to the non-preemptive implementations.



## Acknowledgment

At first I would like to thank my wife Sofia for putting up with all the late nights. I like to thank her for her encouragement and especially for her constant support.

I would like to thank her parents and of course my parents for giving me a place to relax and to feel home. Furthermore, I have to thank for their constant support.

I would like to thank my supervisor Volkmar Uhlig for his guidance and advice. Also I have to thank Dr. Kevin Elphinstone, Gerd Lieflaender and Prof. Dr. Alfred Schmidt who made it possible that I could write this thesis and supported me a lot in doing so.

Finally, I would like to thank the many people that supported me during my studies so far. From them I have to mention in particular the people of the System Architecture Group at the Universität Karlsruhe. Thanks for your support and for the possibility to discuss the many problems that came to my mind during the development of this thesis.

Thanks a lot.



# Chapter 1

## Introduction

Virtual memory management(VMM) is an integral part of operating systems for interactive or partially untrusted applications. In general, this part of the operating system tends to be rather complex. Nevertheless, standard virtual memory policies are often not suitable to serve the requirements of special tasks and applications such as database management (DBMS) [Sto81], garbage collection [AL91] and multi-media applications [Map92].

To fulfill those requirements, two approaches have been proposed:

- *extension* of the VMM-system with application specific policies [BCE<sup>+</sup>94,BSP<sup>+</sup>95, EGM<sup>+</sup>94, CD94, EGK95], and
- *externalization* of virtual memory management to application specific managers [ABB<sup>+</sup>86, RTY<sup>+</sup>87, HC92, KN93, Lie95, Han99].

In micro-kernels, the second approach is applied.

The *Recursive Virtual Address Space Model* [Lie95], proposed for the L4  $\mu$ -kernel, is such an approach of *externalization*. The model provides mechanisms that makes it possible to implement arbitrary virtual memory management and protection policies at user level. Each subsystem can implement those policies, fitting best to the requirements of its applications. It can implement those policies independent of other subsystems. Furthermore, it can use the functionalities provided by other subsystems. This allows to construct systems in a hierarchical manner.

The applicability of the *Recursive Virtual Address Space Model*, however, is currently limited. This is because existing implementations of this model show undesired behaviors or unnecessary complex or slow behaviors such as:

- *long interrupt latencies*,
- *unbounded priority inversion*, or

- *complex timeslice donation or “helping” schemes.*

This thesis investigates techniques and mechanisms to avoid those undesired behaviors. We show that unbounded priority inversion is not inherent to the Recursive Virtual Address Space Model. Instead, we show that it can be avoided by design without having to delay interrupt handling for long times and without having to implement complex timeslice donation or “*helping-schemes*”.

The objective of this thesis is to offer solutions for uniprocessor as well as for small scale multiprocessor systems.

## 1.1 Proposed Solutions

We propose:

- *preemption of long running operations* to avoid long interrupt latencies,
- *post order traversal* to avoid unbounded priority inversion,
- *roll forward combined with scheduler-conscious-locking* to ensure consistency, and
- *restart-point tracking* to guarantee forward progress and to avoid starvation.

To efficiently track the restart point of preempted operations we introduce a new technique: the *token-based preempted-thread list*.

The proposed solutions are evaluated using an implementation in the [L4Ka](#) Pistachio  $\mu$ -kernel.

## 1.2 Outline

Chapter 2 introduces the Recursive Virtual Address Space Model. Furthermore, it explores the problem of unbounded priority inversion in existing implementations. Chapter 3 describes fundamentals applied in this thesis and work, the approach presented in this thesis relates to. Chapter 4 presents the proposed solutions. At first, we focus on solutions for a single hardware page-size. After that, we extend our approach to support multiple hardware page-sizes. Chapter 5 discusses and analyses the experiments. In Chapter 6 we draw conclusions. Furthermore, an outline of future work is given.

## Chapter 2

# Background

This chapter surveys hard- and software mechanisms to implement page-based virtual memory. It introduces the Recursive Virtual Address Space Model and the *mapping database*, a data structure used to implement this model. In addition to that, this chapter surveys small scale multiprocessor systems.

We analyze the problem of unbounded priority inversion and discuss techniques that have been proposed to avoid this problem. One such technique is “helping”.

This chapter ends with identifying the issues and constraints of an implementation of the Recursive Virtual Address Space Model.

## 2.1 Mechanisms for Page Based Virtual Memory

### 2.1.1 Terminology

**Page-Frame** A page-frame (or frame) is a contiguous region of physical memory. It is described by its physical base address and its size, whereby the size is a power of 2 and the base address is aligned to this size.

**Page** A page  $p_v$  is a contiguous region of virtual memory of size  $s$  starting from a virtual base address  $v$ .  $s$  is a power of 2 and one of the hardware page-sizes supported by the processor.  $v$  is aligned to the size  $s$ .

A virtual address-space consists of a mapping that associates each virtual page to a physical page-frame or marks it “non-present”. Additionally, the permitted access to present pages is stored with the mapping.

Page-tables implement this mapping. They are used to translate the virtual addresses of memory accesses to a page into the physical addresses within a frame. Translation lookaside buffer (TLB) hardware caches the result of the most recent translations. The

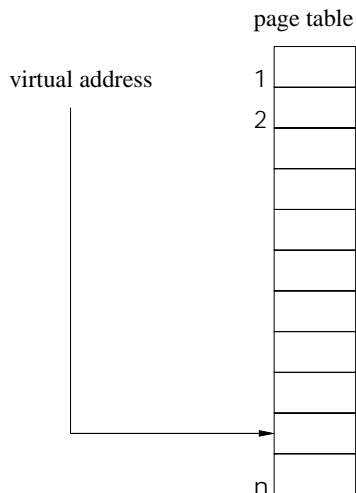


Figure 2.1: *Address translation with a linear page-table*

purpose of the TLB is to speed up subsequent memory accesses to those pages. If no such translation exists in the TLB, i.e. on a TLB miss, the page table is walked by hardware or by software and an entry for the result of this translation is inserted into the TLB.

Several page-table structures have been proposed [JM98, Szm99, LE96, Lie94]. This section gives an overview of the most common.

### 2.1.2 Linear Page-Tables

A linear page-table (Figure 2.1) is an array of page-table entries (PTEs). The virtual address  $v$  to translate is subdivided into two parts. The part with the most significant bits is used to index into the page-table to retrieve a page-table entry. The PTE points to a page frame, or it marks the page as “non-present”. In the first case, the remaining part of the virtual address –  $s$  bits, whereby  $s$  is the page size – is used as an offset into the frame. The second case results in a *page-fault*, a signal to the operating system, that a “non-present” page has been accessed.

Linear page-tables store a page-table entry for all virtual pages of an address space. Often, several of those page-table entries mark the page as “non-present”.

### 2.1.3 Multi-Level Page-Tables

Multi-level page-tables have been proposed to reduce the space required to store page-table entries that mark pages “non-present” in non-populated regions of the address

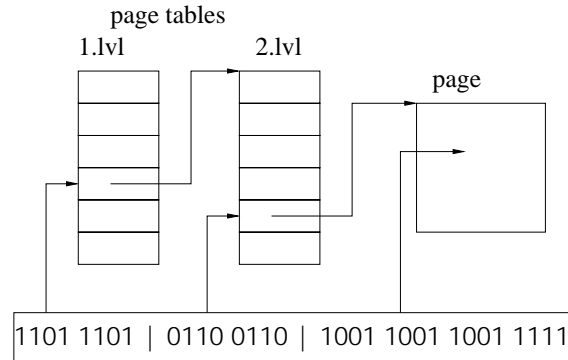


Figure 2.2: Address translation with a multi-level page-table.

space. The multi-level page-table allows for a hierarchy of page-tables to be allocated in multiple levels. Figure 2.2 shows the translation of a virtual address with a multi-level page-table. For the translation of a virtual address  $v$ , the multi-level page-tables subdivides  $v$  into several parts. The translation starts with the part containing the most significant bits. Those bits are used to index into the page table at the first level. The page-table entry found is in one of three possible states:

**page found:** A valid translation of the page is found. The page-table entry contains the physical address of the page-frame and information on the permitted access to it. The bits of the remaining parts offset into the page frame.

**non-present:** The translation stops without translating to a page-frame.

**page-table:** The page-table entry points to a page table at the next level. The bits of the next part are used to index into this page table to find a page-table entry to continue the translation with.

Compared to those parts of the virtual address, the lower-level page-tables are indexed in with, the parts that are used to index into higher-level page-tables contain less significant bits of the address. Therefore, a page-table entry in a lower level stands for a larger region of the virtual address-space than one in a higher level. If no page is mapped in this region, i.e. all page-table entries of higher-level page-tables would mark the pages as “non-present”, the lower-level page-table entry can mark the entire region as “non-present”. In this case, a higher-level page-table is not required. The space this page-table would require is saved. This, however, comes up with the necessity to walk multiple levels of page tables to complete the translation upon a TLB miss.

#### 2.1.4 Guarded Page Tables

To decrease both, the lookup performance of deep multi-level page-tables and the space required for those page-tables in large, sparsely populated address spaces, the guarded

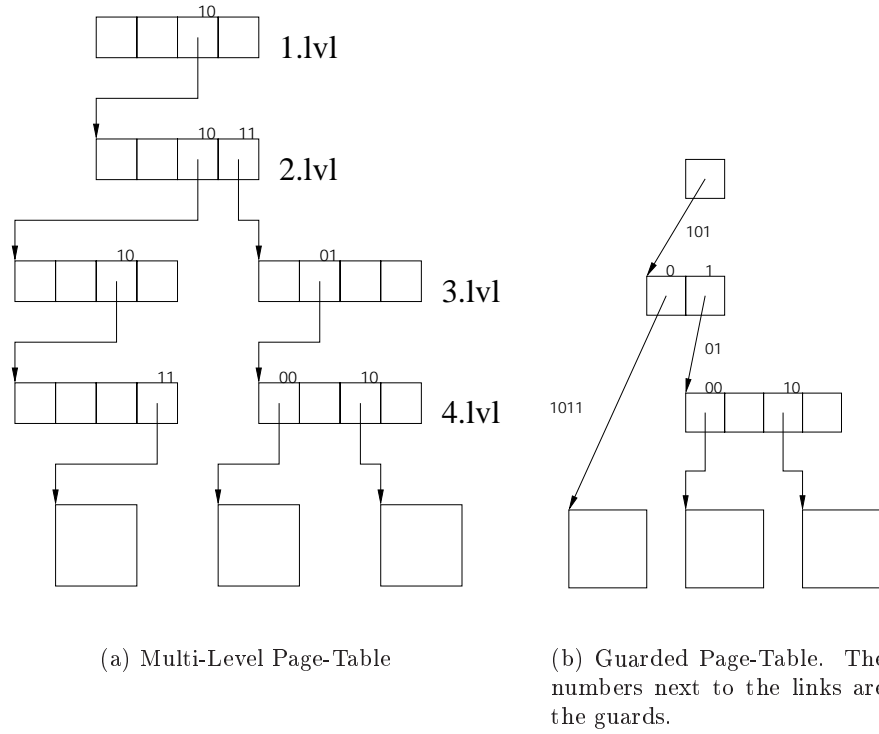


Figure 2.3: Address translation with guarded page-table.

page-table (GPT) [Elp99, Lie94] has been proposed. Furthermore, guarded page-tables allow for efficiently mixing multiple different page sizes.

In sparsely populated address spaces, the page tables of a multi-level page-table often contain only a few page-table entries that point to a page or a page-table at the next level. Guarded page-tables have the ability to compress the translation path through those, sparsely populated regions. Instead of allocating a level of page-table for each fixed sized part of the virtual address, as multi-level page-tables do, the guarded page-table makes it possible to subdivide the virtual address more flexible. Parts containing bits shared by all addresses in a certain region can be split off entirely from the translation. Those bits are stored in the *guards* that directly shortcut to the next level of page tables or to a page. Only with the differing bits, a page-table is indexed. Guarded page-tables allow for differently large page-tables.

We illustrate the benefits of guarded page-tables in the following example: the translating of the following three 10-bit addresses. Figure 2.3(a) shows the translation of

the virtual addresses  $v_1$ ,  $v_2$ ,  $v_3$

$$v_1 = 1011010000$$

$$v_2 = 1011011000$$

$$v_3 = 1010101100$$

by a multi-level page-table, assuming 4 entry page-tables and 4 byte pages. In the first and third level of the multi-level page-table and for  $v_3$  in the fourth level as well, only one page-table entry is used, the rest is marked “not-present”.

In our example (Figure 2.3(b))  $v_1$  and  $v_2$  have a common 6-bit prefix,  $v_1$ ,  $v_2$  and  $v_3$  a common 3-bit prefix. With these common prefixes, the guarded page table shortcuts through the sparsely populated page tables, thereby compressing the translation path. The guarded page-table saves both memory capacity and translation steps in sparsely populated address spaces. Furthermore, it allows for efficiently storing several different hardware page-sizes.

Level- and path-compressed trees (LPC) [Szm99] compress the translation path as GPTs do. Furthermore, level-compression flattens densely populated regions of the address space by replacing complete subtrees with a single page-table. Level compression reduces the depth of the page-table tree in densely populated areas, path compression in sparsely populated areas.

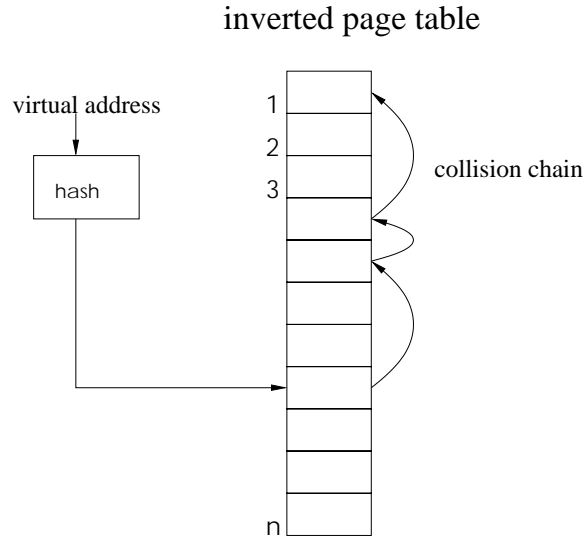
### 2.1.5 Inverted Page-Tables

Instead of storing one page-table entry per virtual page, the inverted page-table (IPT) has one page-table entry per page frame. The page table is called *inverted* because the table is indexed with the physical frame-address rather than the virtual address of the page. Instead, the inverted page-table translates the virtual address by hashing with it into the IPT to retrieve the page-table entry.

Inverted page-tables combine a fast lookup with little space required, provided the hash is collision free. However, in the presence of hash collisions, a collision chain has to be searched for the appropriate page-table entry, or the entire page table has to be rehashed.

## 2.2 Translation Lookaside Buffer

A translation lookaside buffer (TLB) caches the most recent translations from virtual to physical addresses. The entries in a TLB are inserted either by hardware or by software. In the first case, the translation is retrieved by a hardware page-table lookup. In the second, a software handler is responsible to insert the entry.

Figure 2.4: *Inverted Page-Table*

After modifying a translation in the page tables to point to a different frame or when reducing the access rights that are stored with the translation, the corresponding TLB entries need to be invalidated to make the changes to become effective.

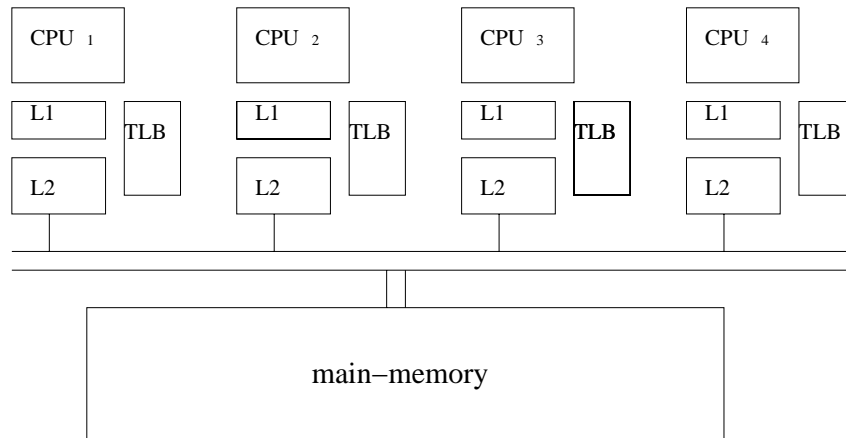
### 2.3 Small Scale Multiprocessors

This section briefly surveys small scale or symmetric multiprocessors. As can be seen in Figure 2.5, symmetric multiprocessor hardware consist of several processors (CPUs) that are connected via a bus or a crossbar to the main memory module. Memory accesses are uniform, i.e. they take equally long for each processor.

Communication between the processors is via the main-memory module, however, a facility exists to deliver an interrupt signal to other processors. This signal is called inter-processor or cross-processor interrupt. Upon receiving this interrupt, the processor executes a handler installed by the operating system.

Each processor has a set of local caches. Those caches are kept consistent by a cache coherency protocol. This protocol ensures, that always up to date values are read from the cache. Modifications to data cached by another processor are written back to main memory, before a read of another processor requests the modified data.

In addition to processor local caches, symmetric multiprocessor systems as well have processor local TLB hardware. However, those TLBs are commonly not kept in a

Figure 2.5: *Symmetric Multi-Processor Hardware*

consistent state. TLB consistency has to be established by the operating system. The process of invalidating non-local TLB entries is called “TLB-shutdown”. Teller [Tel91] and Rosenberg et. al [Ros89] proposed algorithms to shutdown TLB entries.

The synchronization of threads on different processors is supported by symmetric multiprocessor hardware with a series of atomic or memory ordering instructions.

Typically load-store based multiprocessor architectures support an instruction that stores a link between the processor and the memory location it reads (**load-linked**). A conditional store operation tests whether in between the time from loading the memory to storing it, another processor modified the data in memory. If this is the case, the store fails. Otherwise, the memory location is updated atomically and the **store-conditional** operation succeeds.

Other processors typically offer a set of atomic read-modify-write instructions such as: **test-and-set**, **fetch-and-add**, **swap** and **compare-and-swap**.

With those operations, more complex synchronization primitives can be implemented in software.

## 2.4 Recursive Virtual Address Space Model

Traditionally, a virtual address-space is manipulated by modifying the virtual to physical address translations in the page tables. The kernel, however, has to be the only entity in the system, that has direct access to this data structure. Otherwise, protection policies could not be enforced.

On the other hand, implementing virtual memory management and protection policies inside the kernel restricts the system to only a few policies. Those are potentially not optimally suited for any kind of subsystem or application envisaged to run on top of the kernel.

An implementation of virtual memory management and protection policies at user level, makes it possible to implement best suited policies for all subsystems or applications. Direct manipulation of the page tables by user-level memory-managers, however, cannot be allowed. Instead, we have to demand for different solutions to manipulate virtual address spaces outside the kernel.

To securely manipulate virtual address spaces at user level, Liedtke has proposed a recursive construction-scheme of virtual address spaces: the Recursive Virtual Address Space Model.

The key idea of this model is to construct an address space recursively by providing it access to the memory of the constructing address space. The important restriction is, that access may only be given to those page frames, the constructing address space itself has access to. To achieve this, virtual pages are used as parameters for those operations modifying address spaces. The frames operated on, are retrieved by looking up the page tables of the source address-space.

The recursion is initialized by constructing an initial address space called  $\sigma_0$ . This address space has an idempotent mapping to all physical memory, except to those required by the kernel.

The Recursive Virtual Address Space Model allows to construct and manipulate address spaces with the following three operations: *map*, *grant* and *unmap*. We refer to those operations as *address space modifiers*.

### 2.4.1 Map

Any thread of an address space – the mapper – can *map* any of its pages to another address space, provided that a thread in the target address space – the recipient – agrees. Afterwards, the mapped page frames are accessible in both address spaces (see Figure 2.6).

It is important to note, that access rights cannot be elevated. The page is mapped into the target address space with the minimum of the rights specified by the mapper and the rights the mapper possesses itself.

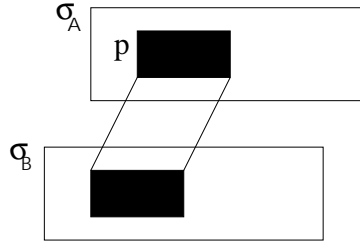


Figure 2.6: A page  $p$  is mapped from address space  $\sigma_A$  to  $\sigma_B$ . Afterwards,  $p$  is accessible in both address spaces.

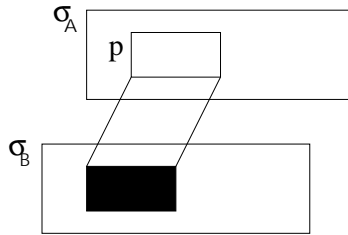


Figure 2.7: A page  $p$  is granted from address space  $\sigma_A$  to  $\sigma_B$ . Afterwards,  $p$  is accessible in  $\sigma_B$ , but no longer in  $\sigma_A$ .

### 2.4.2 Grant

As with *map*, any thread of an address space can *grant* any of its pages to another address space, provided an agreement of the recipient. The granted pages are removed from the granter's address space and included into the target address space (see Figure 2.7).

Again the pages are made accessible in the target address space with the minimum of the access rights specified and the access rights the *granter* had to the page, prior to removal.

### 2.4.3 Unmap

Any thread of an address space can *unmap* any of its pages. Afterwards, the access rights, specified in the *unmap* operation, are revoked from all address spaces that directly or indirectly received the page from this thread or from any thread in the same address space (Figure 2.8). A revocation of all access rights from a page results in removing that page completely <sup>1</sup>.

---

<sup>1</sup>Note, the *unmap* operation guarantees, that the access rights are revoked by the time, the *unmap* operation returns. It is up to the implementation, when exactly the rights are revoked during the *unmap* operation, as long as it does not return from the *unmap* operation as long as a derived mapping

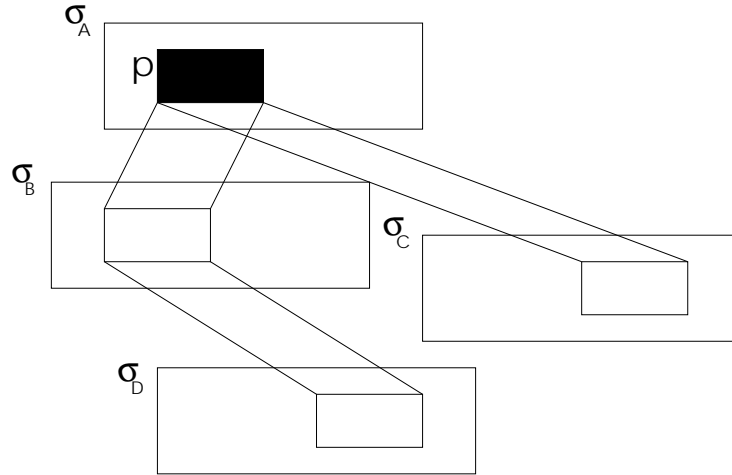


Figure 2.8: A page  $p$  is unmapped from all address spaces that directly ( $\sigma_B$ ,  $\sigma_C$ ) or indirectly ( $\sigma_D$ ) received it from  $\sigma_A$ . Afterwards,  $p$  is no longer accessible in  $\sigma_B$ ,  $\sigma_C$  and  $\sigma_D$ .

*Unmap* does not require explicit agreement of the address spaces the pages are revoked from. Nevertheless, the operation is safe because it is restricted only to owned pages. Upon accepting a page mapped or granted, the receiving thread, implicitly agrees on a potential revocation.

With a special case of *unmap*, a thread can revoke access from the pages in it's address space as well. We call this special case *inclusive unmap*.

The number of pages to be revoked with *unmap* cannot be controlled by the thread executing the *unmap* operation. It can control the number of direct mappings – those pages are mapped by threads in it's address space. However, it has no control on the number of indirect mappings because with receiving a page, the receiver gets as well the right to go on *mapping* it <sup>2</sup>.

---

exists that still has an access right that was revoked. By the time *unmap* returns, no derived mapping has an access right that was revoked by the *unmap* operation.

<sup>2</sup>The Recursive Virtual Address Space Model does not limit the number of derived mappings. One way to extend the model is to add a quota to each page mapped. In the following we briefly present this solution. However, a complete discussion of appropriate mechanisms to limit the number of derived mappings is out of the scope of this thesis.

The number of derived mappings can be limited by adding a quota to each page mapped. This quota limits the number of mappings that can be directly or indirectly derived from this page. As with access rights, the quota of the owned page limits the number of direct mappings and the quotas handed out. More precisely, the sum of all quotas mapped plus the number of derived mappings has to be less or equal to the quota received with the page. Restricting allowed values for the quota to 0 and infinity

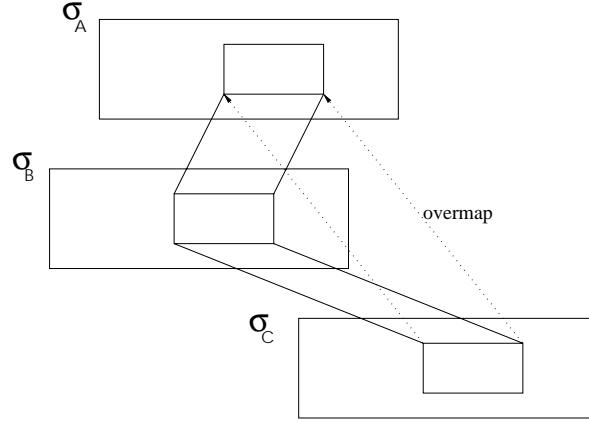


Figure 2.9: The conflicting page  $p$  is implicitly unmapped before it is overmapped. This leads to  $p$  being completely removed in the address spaces  $\sigma_A$ ,  $\sigma_B$  and  $\sigma_C$ . The map operation of  $\sigma_C$  results in a “no-op”

The *unmap* operation has to revoke access from all address spaces that directly or indirectly received the mapping from a thread in the address space of the unmapping thread. This number can be large, leading to a long running *unmap* operation, and the unmapping thread has no way to limit this number. When executing this operation non-preemptively, no other thread on this processor can preempt the *unmap* operation. Because this includes those threads participated in interrupt handling, interrupt latency increases by the time it takes to complete this long running *unmap*. To avoid those long interrupt latencies, we propose a preemptable implementation of *unmap*.

#### 2.4.4 Overmap and Fast Overmap

Prior to the *mapping* or *granting* of a new page we check whether this new page conflicts with pages, already mapped in the mappee’s address space. In case of a conflict, we implicitly perform an *inclusive unmap* on the old pages.

This operation is called *overmapping*.

*Overmapping* avoids cyclic mappings as can be seen in Figure 2.9. The implicit *inclusive unmap* of the destination area removes the source page that if *mapped* / *granted* would close the cycle. The resulting *map* / *grant* operation is a *no-op*.

The *inclusive unmap* on *overmap*, removes the *overmapped* pages from all address spaces that directly or indirectly received it from the mappee. In some situations, however,

---

results in the quota being a *map-right*. The issues in implementing this quota-based-scheme are: the bookkeeping and enforcement of the quotas, the choice of appropriate mechanisms to publish the quota to threads in the address space and the choice of how to react when the quota is exceeded. Further work is required to explore this and other mechanisms to limit the number of derived mappings.

this is both unnecessary and too slow. In those cases, a way to increase access to a page, avoiding the *unmap*, is preferable. This special case of *overmapping* is called *fast overmapping*.

### Fast Overmap

*Fast overmap*<sup>3</sup> is a special case of *overmapping*, avoiding the implicit *unmap*. Access rights can be extended if the very same mapping already exists. This is the case, when:

- the mapper maps from the same address space as the existing mapping,
- the mapper maps from the same virtual source address as the existing mapping, and
- the mapper maps to the same virtual destination address as the existing mapping

### 2.4.5 Page Reference Information Retrieval

Some processors, for example Intel's IA32 architecture [Int02], set reference information in the page tables – i.e. they store whether a page has been accessed, modified, or the code in it has been executed. Babaoglu et. al [BJ81] proposes an algorithm simulating reference bits on architectures without hardware implemented reference information.

Direct access to the page tables cannot be given to user-level applications or subsystems, since otherwise, implementing protection would be impossible. Therefore, the relevant question is how to provide the subsystems implementing memory management and protection policies with page reference information.

Possible solutions are:

- *not providing page reference information*  
The Recursive Virtual Address Space Model does not provide page reference information. Instead, a similar algorithm as proposed by Babaoglu et. al has to be applied to simulate reference information when mapping the page.
- *a system-call reads page reference information*  
To retrieve the information whether the page was referenced (R), written (W) or executed (X) by any thread of an address space that directly or indirectly received the page, a system-call reads and resets the page reference information. Because of the required traversal of all derived mappings, we combine this system-call with the *unmap* operation. Upon traversing the pages, access is to be revoked from, the *unmap* operation reads and resets the referenced bits from the page tables

---

<sup>3</sup>*Fast overmap* and *Reference information retrieval* are extensions to the recursive virtual address space model.

and returns the result. *Unmap* can be configured to revoke no right. In this case, only the reference information is read and reset.

In a deep hierarchy of subsystems, the first solution may lead to high overhead due to repeatedly unmapping and remapping to implement reference information. The trade-offs of *page-reference-information retrieval* are the overhead added to *unmap*, reading the reference bits, compared to the costs of the additional page faults generated.

### 2.4.6 Multiple Hardware Page-Size Support

Talluri et. al [TKHP92] investigated the tradeoffs of supporting different hardware page-sizes in the TLB hardware.

Mapping a large memory object with large pages into an address space increases TLB coverage. When accessing the large object, fewer TLB entries have to be assigned to cover the entire memory object. The chance of a TLB hit increases as well, because a larger area is covered by the large page. The likelihood of having a valid translation cached from a previous access increases.

On the other hand, allocating the memory object in physical memory such, that it can be mapped in larger pages is more difficult. A large, free page frame needs to be found. A second counter argument is the write back granularity. The assignment of the memory object with larger pages requires to write the memory object back to secondary storage in a larger granularity, or the larger object has to be split up into a set of smaller pages and reconcatenated later on. Nevertheless, selecting the appropriate hardware page-sizes for the appropriate memory objects and applications may increase overall system performance compared to a system with a single hardware page-size.

### Flexpages

Hohmuth et. al [HWL96] propose *flexpages* (fpages) to generalize from a distinct hardware page-size. Similar to a page, a flexpage is specified by its virtual base address and by its size, whereas the size is a power of two and at least the smallest supported hardware page-size. The virtual base address is size aligned.

The Recursive Virtual Address Space Model applies fpages to specify both, the pages to be *mapped*, *granted* or *unmapped*, and the region of the virtual address space to accept mappings in (see the L4-User Manual [AH99] for more details).

The following three special cases can occur for the *address space modifiers*: *map*, *grant* and *unmap*, when operating on multiple hardware page-sizes: *split mappings*, *united mappings* and *partial unmap*.

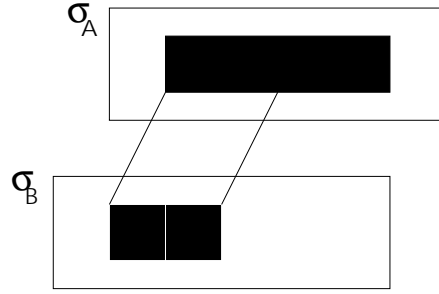


Figure 2.10: *Split mapping out of a large page.*

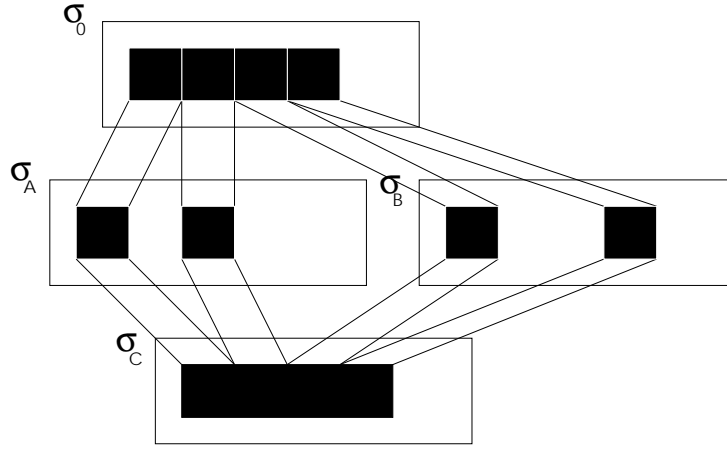


Figure 2.11: *Adjacent mappings are united into a large page.*

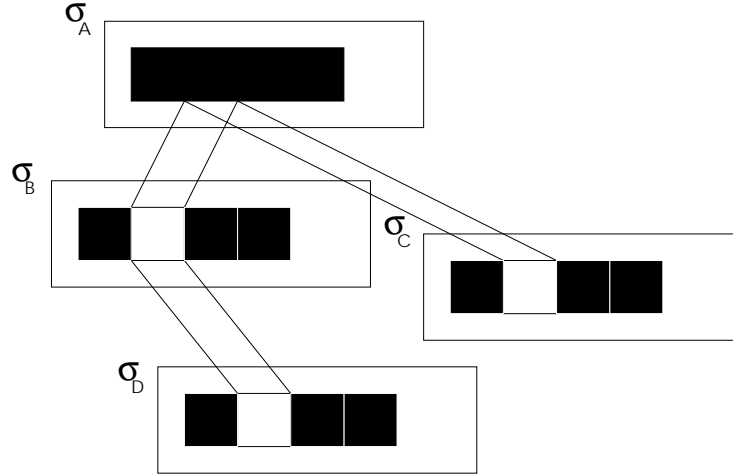
### Split Mappings

*Split mappings* (Figure 2.12) can result from a *map* (or *grant*) operation if the fpage to be mapped covers only a part of a larger page in the mapper's address space. The fpage is mapped with smaller page sizes into the target address-space, however, those page sizes still have to fit into the fpage.

### United Mappings

*United mappings* (Figure 2.11) unite several adjacent pages in the target address-space into a large page. The pages as well as the according page frames have to be contiguous in memory and must have been mapped with the same access rights.

*United mappings* are not evaluated in this thesis.

Figure 2.12: *Partial unmap of a large page.*

### Partial Unmap of Large Pages

Revocation of a part of a large page is called a *partial unmap*. Therefore, the large page is split up into smaller pages.

Alternatively, instead of splitting up the large page into smaller pages, the entire large page could be revoked.

*Uniting mappings* of pages mapped from different threads requires to split up large pages. Otherwise, one thread can revoke pages mapped by another thread. Guarantees concerning the presence of the page, the second thread associates with the page, can be broken by the first. An example for this is the guarantee, not to revoke the page for a certain time, i.e. to pin the page.

Figure 2.13 illustrates this example: A thread  $\tau_A$  maps a page  $p_v$  and guarantees not to unmap it for a certain period of time –  $p_v$  is pinned.  $\tau_B$  maps  $p_{v+1}$  adjacent to  $p_v$  and both  $p_v$  and  $p_{v+1}$  are united into  $p'_v$ . A revocation of  $p_{v+1}$  resulting in revoking  $p'_v$ , i.e. the page  $p_v$  as well, breaks the pinning guarantee.

### 2.4.7 Fundamental Consistency Requirements

The following consistency requirements have to be met in any implementation of the Recursive Virtual Address Space Model.

**revokability:** The revocation of pages  $p$  directly or indirectly mapped into an address space  $\sigma_i$  has to be possible at any time.

Otherwise, the page cannot be revoked, for example to write it back to secondary storage or to hand it out to another client.

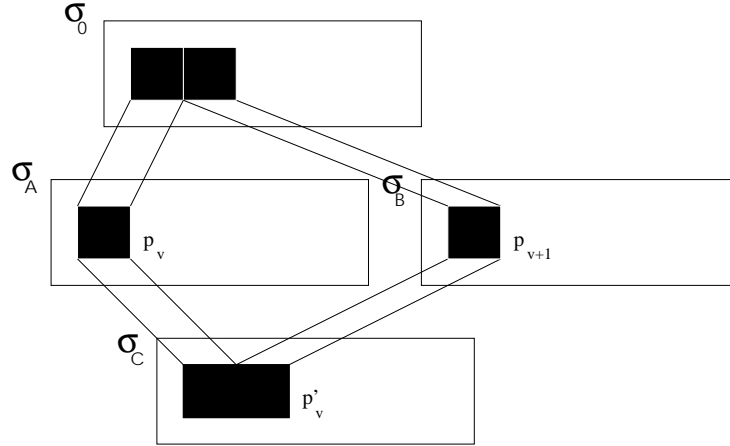


Figure 2.13: *Revocation of a united page with different guarantees associated to its parts.*

**restartability:** Aborted and not fully completed operations modifying the address space mapping, have to be restartable.

**no privilege elevation:** The receiver of a page  $p$  must not be able to elevate the access rights it received  $p$  with. The only way to increase the access rights of  $p$  is to request a new mapping of  $p$  with increased rights. This implies that the mapper, that maps the new mapping, itself has sufficient rights.

If a receiver of a page  $p$  would be able to extend its access to a page, protection policies cannot be enforced.

## 2.5 Mapping Database

The *Recursive Virtual Address Space Model* proposes a recursive construction of address spaces with *map*, *grant* and *unmap*. With those operations, all subsystems can implement their own virtual memory management and protection policies independent of other – even untrusted – subsystems. The operations itself, however, have to be implemented by an entity trusted by all subsystems, i.e. the kernel.

*Map* and *grant* can be implemented solely by modifying the page tables. *Unmap*, however, requires additional information to revoke access from all directly or indirectly derived mappings of a page  $p$ . It needs the virtual addresses and the address spaces the page  $p$  has been mapped to. This information is stored in a data structure referred to as the *mapping database*.

The *mapping database* contains an entry  $m$  for each valid page-table entry of an address space  $\sigma_i$ , i.e. for each page-table entry that translates a page  $p_v$  into a page frame. Given

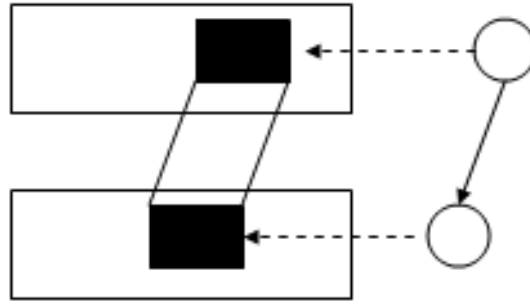


Figure 2.14: *Map* inserts a new node into the mapping-database.

this entry  $m(p_v, \sigma_i)$ , the purpose of the *mapping database* is to find the entries of all virtual addresses and address spaces,  $p_v$  has been mapped to. *Unmap* iterates through those *mapping-database* entries and revokes the access from the corresponding pages.

### 2.5.1 Structure of the Mapping Database

A page  $p_v$  can be mapped several times to one single address space as well as to many address spaces. When *uniting* adjacently mapped pages into a single large page, a page  $p'_v$  can originate from several pages  $p_{v_i}$ . Therefore, the *mapping database* is a graph. The graph is directed because pages are always mapped from a source address-space into a target address-space. It is acyclic because *overmapping* avoids cyclic mappings.

The nodes  $m(p_v, \sigma_i)$  in this acyclic directed graph are the mapping-database entries – or in the following *mappings*. They denote an address space  $\sigma_i$  and a virtual address of page  $p_v$ . The mapping node stores sufficient information to find and modify the page-table entry of the page  $p_v$  and to invalidate the corresponding entries in the TLBs of the processors in the system.

Outgoing edges are linked to nodes representing pages and address spaces that directly received the mapping of  $p_v$ . Inbound edges are linked to nodes,  $p_v$  originates from.

The root nodes of the graph are the nodes, denoted to the initial address-space  $\sigma_0$ . Because of  $\sigma_0$  is constructed with an idempotent mapping of virtual to physical addresses, the  $\sigma_0$ -nodes directly relate to the page frames.

- *Map* inserts a new node for each page mapped into the mapping database graph and links it as a child to the mapping node of the source page (Figure ??).
- *Grant* does not modify the mapping database structure, but modifies the page and address space information in the mapping node (Figure 2.15).

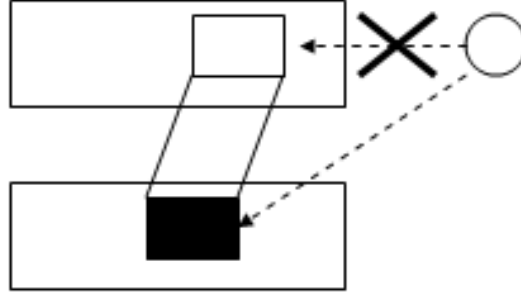


Figure 2.15: *Grant* modifies the existing mapping node only.

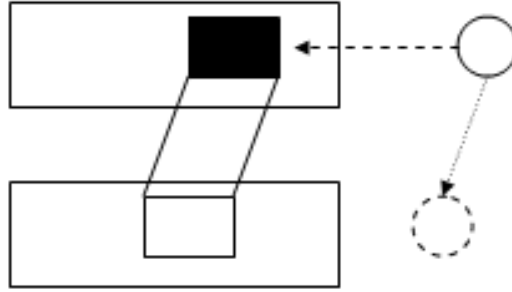


Figure 2.16: *Unmap* removes a node from the mapping database if all rights are revoked from the page.

- *Unmap* iterates through the subgraph of derived mappings to revoke access from the corresponding pages. If all access is revoked from a page, the corresponding mapnode is removed from the mapping database graph as well (Figure 2.16).

With the exception of *united mappings*, each page originates from exactly one other page (except the pages of  $\sigma_0$ ). If mappings are not *united*, the indegree of all nodes in the mapping database is 1, except for the root mapping nodes of  $\sigma_0$ . The resulting mapping-database structure is a tree.

## 2.5.2 Requirements and Constraints

### Interface to the Mapping Database

The *mapping database* offers the following operations as an interface to the database for the *address space modifiers*: *map*, *grant* and *unmap*.

**find\_map\_node (virtual\_address, address\_space)** Given the virtual address and the address space of a page, **find\_map\_node** searches for the corresponding mapping node in the database.

**find\_page\_table\_entry (mapnode)** **find\_page\_table\_entry** returns the corresponding page-table entry given a mapnode.

**insert\_mapping (parent\_mapnode)** Inserts a new mapnode as a child of the given parent mapnode.

**remove\_mapping (mapnode)** Removes the mapnode from the mapping-database graph.

**Iterator (root\_mapnode)** The *mapping database* provides *unmap* with an iterator interface to iterate through all mapnodes of mappings, directly or indirectly derived from the root mapnode.

**find\_first\_leaf (root\_mapnode)** The **find\_first\_leaf** operation searches for the left most leaf node of the subtree with root node: **root\_mapnode**.

This operation is required only for a post order traversal of *unmap*. The traversal path starts at this node. Reasoning for post order traversal is given in the next section.

### Constraints

The following objectives have been committed for the design and implementation of the *mapping database*.

**space:** The *mapping database* should require as little space as possible.

**time:** In general the *address space modifiers* should perform best possible.

**preemptability:** Long running operations on the *mapping database* should be preemptable to avoid long interrupt latencies.

**recursion:** The limited stack size of an in kernel implementation of the mapping database, demands for iterative instead of recursive algorithms.

**consistency of page tables and mapping database:** Modifications of page tables and mapping database have to be consistent. In particular, the three consistency requirements of the *Recursive Virtual Address Space Model* have to hold: *revokability*, *restartability* and *no privilege elevation*.

**bounded priority inversion:** Unbounded priority inversion has to be avoided by design of the mapping database.

**no starvation and deadlocks:** The mapping database has to be free of starvation and must not lead to a deadlock.

**uniprocessor and symmetric multiprocessor systems:** The objective of this thesis is to provide solutions both for uniprocessor as well as for symmetric multiprocessor systems.

## 2.6 Unbounded Priority Inversion

Existing implementations of the *Recursive Virtual Address Space Model* have lead to undesired behavior or unnecessary complex behavior such as:

- unbounded priority inversion, or
- long interrupt latencies

Complex helping-schemes have been applied to avoid both.

This section examines circumstances leading to unbounded priority inversion in existing implementations of the Recursive Virtual Address Space Model and the means that they apply to avoid that undesired system behavior.

### 2.6.1 Definition

Priority inversion is the phenomenon that the execution of a higher prioritized thread is prevented by a lower prioritized thread [SRL90]. Priority inversion is unbounded if the time of priority inversion is not bounded.

Priority inversion happens if a higher prioritized thread requires a resource held by a lower prioritized thread. The higher prioritized thread is prevented from execution until the lower prioritized releases this resource.

The mapping database is such a resource for the address space modifying operations of the Recursive Virtual Address Space Model: *map*, *grant* and *unmap*.

### 2.6.2 Unbounded Priority Inversion in Address Space Construction

*Map*, *grant* and *unmap* have to prevent conflicting operations from execution, while they modify the mapping database and the page tables. Otherwise, the mapping-database and page-table consistency cannot be guaranteed. If the threads of those prevented operations are higher prioritized as the thread that prevents them, priority inversion occurs. Unbounded priority inversion occurs if the time the higher prioritized threads are prevented from execution is not bounded.

In this thesis orthogonality of scheduling and address space construction is assumed. The Recursive Virtual Address Space Model should not influence scheduling decisions – in particular priority assignment. Conversely address space construction should work independently of the priorities of the participating threads. In particular, a mapper should be able to map to a mappee of any priority, i.e. whether this mappee has a lower or higher priority than the mapper. In any case, the mapper must be able to *unmap*

this page at any point in time.

It is particularly complicated to bound the time of priority inversion, if a preempted operation prevents higher prioritized operations from execution. For example by holding a lock. This is because we cannot assume, that the lock-holder is scheduled again to release the lock.

In existing implementations two solutions have been applied to solve this problem. The first is to roll forward the operation to completion. The second solution is to “help” out the operation that blocks higher prioritized operations.

Mapping database operations, *mapping*, *granting* or *unmapping* multiple pages can be preempted after completing with a single page. The Recursive Virtual Address Space Model does not require to order those operations. A multi-page operation therefore can be seen as multiple single-page operations.

*Mapping* or *granting* a single page to a target address-space requires to modify only a few nodes in the mapping database. *Map* inserts a new node, *grant* modifies the node of the granted page. Those operations can be rolled forward to completion without delaying interrupt handling for a long time. A potential priority inversion caused by those operations is bounded because they are non-preemptively executed to completion. A single page *unmap*, however, has to revoke access from a potentially large number of pages of the derived mappings. A roll forward of the entire *unmap* operation may result in long interrupt latencies. Therefore, *unmap* should be preemptable. To avoid inconsistencies while processing a single node, we assume, that *unmap* is rolled forward for the time it takes to process this node. Even then, *unmap* might cause unbounded priority inversion as it is shown in the next section.

### 2.6.3 Unbounded Priority Inversion of Preorder Unmap

Common to all existing implementations of the *Recursive Virtual Address Space Model* is a preorder traversal of the mapping database tree for the *unmap* operation. Unbounded priority inversion occurs, when a lower prioritized *unmap* operation prevents a higher prioritized mapping database operation from execution by holding a resource required by the higher priority operation. Note, a lower prioritized single page *map* and *grant* cannot cause unbounded priority inversion if rolled forward to completion. Because of the roll forward, the time is bound that *map* or *grant* potentially blocks the *unmap* operation.

Even if the lower prioritized *unmap* can be implemented to release all its resources before being preempted, unbounded priority inversion may occur in the presence of a higher prioritized *unmap*. The following example illustrates this situation (Figure 2.17):

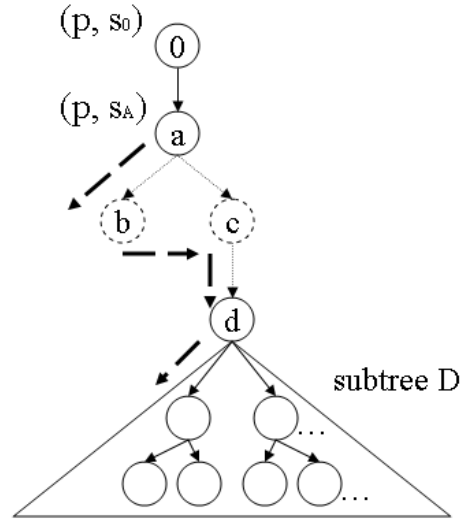


Figure 2.17: *Situation of unbound priority inversion of concurrent unmap operations. The dashed lines mark the traversal path of the unmap operation.*

We assume that the *unmap* operation can be preempted after operating a single mapping node, i.e. after revoking access from the page that corresponds to this mapping node. At this time, all resources held by *unmap* are released. Further, we assume a pre-order traversal of the subtree *D* of the mapping database to process.

A page *p* is mapped from address space  $\sigma_A$  to  $\sigma_B$  and  $\sigma_C$ .  $\sigma_C$  further mapped it to  $\sigma_D$  and  $\sigma_D$  continued mapping the page, resulting in a large subtree with root map node *d*.

A thread  $\tau_A$  in address space  $\sigma_A$  starts to unmap *p*.  $\tau_A$  manages to remove  $p_B$  in  $\sigma_B$  and  $p_C$  in  $\sigma_C$  before it is preempted. Next, a higher prioritized thread  $\tau_C$  in address space  $\sigma_C$  unmaps  $p_C$ .

Because the mapnode *c* has already been removed from the mapping database,  $\tau_C$  is no longer able to revoke the subtree *D* with root-mapnode *d* directly. Instead it has to wait for  $\tau_A$  to be rescheduled and complete its operation. Priority is inverted. The priority inversion is unbounded because we do not require the scheduler to reschedule  $\tau_A$  in a limited time again.

The following solutions to avoid unbounded priority inversion in this scenario have been implemented:

1. Unbounded priority inversion is avoided if we would immediately return from  $\tau_C$ 's *unmap* operation, i.e. not waiting for  $\tau_A$  to complete. This, however, violates the revokability requirement, since *D* can no longer be revoked by  $\tau_C$ .

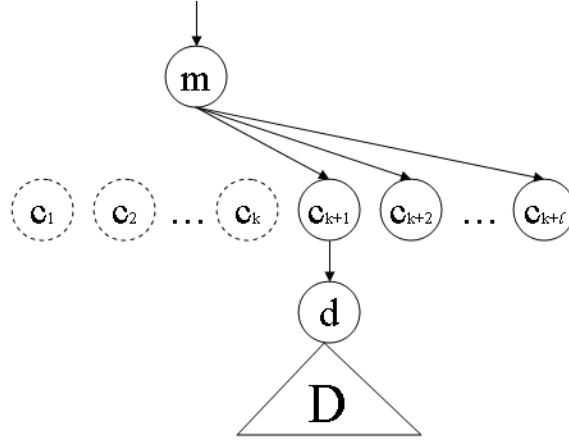


Figure 2.18: *Generalization of the situation of unbound priority inversion of concurrent unmap operations to other traversal methods.*

2. In the Fiasco  $\mu$ -kernel the thread  $\tau_C$  donates its timeslice to  $\tau_A$  to help it out of the unmap operation, i.e. to complete the processing of the subtree  $D$ .
3. Other implementations (for example the x86 assembler implementation L4-Orangepip) execute  $\tau_A$ 's unmap operation to completion, i.e. unmap is implemented non-preemptively. This may result in long interrupt latencies if the subtree  $D$  is large.

If instead of  $\tau_C$ , a thread  $\tau_B$  in  $\sigma_B$  would unmap  $p_B$ , the problem does not show up. This is the case because there are no derived mappings from the mapping node  $b$ . The fundamental difference, causing the problem for  $\tau_C$ , but not for  $\tau_B$ , is that the preemption of  $\tau_A$  leaves behind the subtree  $D$  of derived mappings. This subtree  $D$  cannot directly be reached by  $\tau_C$ , and it is large, so that rolling forward  $\tau_A$ 's operation leads to long interrupt latencies.

#### 2.6.4 Generalization

*Unmap* has to iterate through all mapnodes  $m(p_v, \sigma_i)$  of the subgraph of derived mappings. For each of those nodes processed, access is revoked from the page corresponding to it by modifying the page-table entry. For an arbitrary traversal through the tree, the following condition holds: Before processing the mapping node  $m(p_v, \sigma_i)$ , i.e. before revoking access from the page  $p_v$ , access from the pages of  $k$  of its child nodes has been revoked before,  $l$  child nodes remain to process afterwards (see Figure 2.18).

In a pre-order traversal of the tree,  $k = 0$  for all mapnodes. All child nodes are processed after the parent node  $m$  has been processed.

It is possible to construct a scenario with a similar effect as described above for any traversal order that has a node  $m(p_v, \sigma_i)$  with  $l > 0$ . The constraints for this scenario

are that a subtree similar to the subtree  $D$  exists that is derived from  $m$ . This subtree has to be sufficiently large to cause long interrupt latencies when rolling forward the *unmap* operation. Furthermore, it has to be linked to one of the  $l$  child nodes that have not been processed yet. After revoking  $m$ , this subtree  $D$  is no longer reachable by a thread in  $\sigma_i$ . Unbounded priority inversion does occur if the priority of the preempted *unmap* is lower than the priority of the thread in  $\sigma_i$ , *unmapping*  $m$ .

A traversal order with  $l = 0$  guarantees, that all derived mappings in the tree remain reachable. This is because the following condition holds for all mapping nodes: before revoking access from the page of node  $m$ , all child mappings and grant child mappings have been processed before. The traversal method with  $l = 0$  for all mapping nodes is called *post order traversal*.

### 2.6.5 Helping

The technique of helping [SRL90,HH01] has been proposed to avoid unbounded priority inversion. A lower prioritized thread blocking a resource needed by higher prioritized threads is “helped” to release the resource and therefore to bound the time, priority is inverted.

This is accomplished by raising the priority of the lower prioritized thread to the priority level of the waiting threads for the time, it holds the resource. The lower prioritized thread effectively runs on the priority level of the waiting thread with the highest priority.

The two most commonly used helping protocols are

- the *priority inheritance protocol*, and
- the *priority ceiling protocol* [SRL90].

#### Priority inheritance protocol

The key idea of the priority inheritance protocol is that a thread  $\tau_j$  inherits the highest priority of the threads it blocks by holding a shared resource. The priority of  $\tau_j$  is raised to that of the highest prioritized thread requesting the resource. When  $\tau_j$  releases the resource it unblocks the highest prioritized thread it has blocked. After releasing the resource,  $\tau_j$ 's priority is decreased to its original level.

The priority inheritance protocol bounds the time, priority is inverted. However, it does not prevent deadlocks. Because of chain blocking, a higher prioritized thread may have to wait for a long time until it gets the resource.

### Priority ceiling protocol

The priority ceiling protocol has been proposed to avoid chain blocking. The key idea is to add a ceiling to each resource. This ceiling is the highest priority of all threads currently blocked on the resource. A thread  $\tau_j$  is allowed to acquire and block a released resource, if its priority is higher than the ceiling.

When applying the priority ceiling protocol, the highest prioritized thread blocking on the resource has to wait at most until the current holder releases it. The priority ceiling protocol prevents deadlocks.

### Wait-free locking with helping

Hohmuth et. al [HH01] proposes a wait-free, i.e. non-blocking starvation free, locking-with-helping scheme. Each resource is protected by a lock. The lock is complemented by a helper stack. When a thread  $\tau_B$  requests the resource and finds it locked by a thread  $\tau_A$ , it inserts its thread control block on the top of the helper stack. Next, it helps  $\tau_A$  to free up the resource and to release the lock by donating its timeslice to it.  $\tau_A$ , when releasing the lock, donates the resource to the thread that is at the top of the helper stack.

Compared to donating priorities along a FIFO wait queue, the stack guarantees a LIFO processing of the waiting threads. When requiring that only one thread exists per priority level and that threads are scheduled according to hard priorities, the highest prioritized thread lands at the top of the stack. Because this thread does not release the processor voluntarily after entering the stack but donates its entire timeslice to the lock-holder, it cannot be preempted by a lower prioritized thread. When we further require, that the lock-holder does not voluntarily release the CPU and does not block, starvation is avoided.

In the L4  $\mu$ -kernel, however, those requirements are not given. First, multiple threads with the same priority may exist. One of those might have entered the stack first, but may be prevented from ever getting the lock because other threads at this priority level can get in front of it on the stack. Starvation occurs. Second, L4 supports hand off scheduling by allowing for a thread to voluntarily donate its timeslice to another one. With this, a higher prioritized thread can temporarily increase the priority of a lower prioritized thread by donating its timeslice. If this lower prioritized thread attempts to acquire the lock while its priority was boosted, it gets on the stack in front of a higher prioritized thread. Again, starvation may occur if this situation persists. Freeness of starvation cannot be guaranteed with the *wait-free locking with-helping scheme* in the L4  $\mu$ -kernel without restricting possible scheduling policies and timeslice donation.

To “help” out the lock-holder, the timeslices of the “helping” threads are implicitly

donated to the lock holder. The scheduler, however, accounts this time to the threads that donated it instead of accounting it to the lock-holder. To precisely account for donated time, the kernel has to follow the chain along which the timeslice is donated to the lock holder. Even in the stack based approach, it has to account donated time to the lock holder and when this releases the lock, it has to account the remaining timeslice to the next thread on the stack. Furthermore, it has to provide this information to the scheduler. Both, the accounting and to inform the scheduler about the results is complicated to implement. A more direct approach, for example to roll forward the lock holder is easier to account.

In a multiprocessor system, the scheduler decides on both, the amount of time a thread is allowed to execute and the processor it executes on for that time. Donating this CPU-time to a thread on another processor, therefore requires to either migrate the target thread, or to allow for the donated time to be executed on the other processor later on. Not all scheduling policies tolerate the second option.

Having to implement a “helping” scheme for those, therefore requires to migrate either the lock holder or the “helper” to the same processor, such that the “helper” can donate the timeslice it gets to the lock holder to “help” it release the lock.

Hohmuth et al. proposes a priority queue instead of the stack when migrating the lock-holder or when timeslices can be donated across processor boundaries.

When considering the resident cache working set of the lock holder or of the helper, frequent migrations and in particular the transfer of this cache working set may be costly operations. In this situations, having to migrate the lock holder or the helper to be able to donate the timeslice of the helper may be both too complex to implement and too slow.

The solutions presented in this thesis attempt to avoid the necessity of having to implement “helping”.

## 2.7 Summary

This section surveyed the necessary background information to understand the *Recursive Virtual Address Space Model*. The important property of this model is that address spaces can be constructed at user level. Therefore, the model proposes the three *address space modifiers*: *map*, *grant* and *unmap*.

We presented two optimizations: *fast overmap* and *reference-information retrieval* with *unmap*. Multiple hardware page-size support can lead to *split mappings*, *united mappings* and *partial unmap* of large pages.

The mapping database stores directly and indirectly derived mappings to allow a revocation via *unmap*. It is the fundamental data structure in the implementation of the

*Recursive Virtual Address Space Model.*

The problem of unbounded priority inversion in the existing implementations of this model has been analyzed. We identified *post-order traversal* of the mapping-database subtree to process on *unmap* as a technique to avoid unbounded priority inversion. In combination with rolling forward parts of the operations that modify address spaces, this traversal method avoids the necessity to implement “helping”. The problems and difficulties of those “helping-schemes” have been presented.

## Chapter 3

# Fundamentals and Related work

This chapter provides a review of related work and surveys the fundamentals this work is based on.

In this thesis, we will not introduce to tree and graph theory, but assume the reader is familiar with its terminology. Please refer to [Knu97, Section 2.3, pp 308ff] for more details.

### 3.1 Existing Implementations of the Recursive Virtual Address Space Model

Liedtke [Lie95] proposed the Recursive Virtual Address Space Model and first implemented it in the L4  $\mu$ -kernel. Since then, slightly differing variants have been implemented in the different kernel versions and ports.

#### 3.1.1 The $\mu$ -kernel Approach

The  $\mu$ -kernel [ABB<sup>+</sup>86,KN93,ARS89,Lie95,Hil92] is one approach to decrease the complexity of todays operating systems. The key idea is to externalise any policy from the kernel [Lie96] into protected servers executed at user level.

While in  $\mu$ -kernels of the first generation such as Mach, Spring, Chorus only few operating system concepts were deferred into user-level servers, such as paging for example, the second generation  $\mu$ -kernels such as L4, were build on the concept of minimality. Only those functionalities are accepted in the  $\mu$ -kernels, that cannot be implemented in user-level servers.

### 3.1.2 L4 $\mu$ -kernel

The L4  $\mu$ -kernel was developed at the Universität Karlsruhe, at IBM and at the GMD. It provides only three abstractions: threads, address spaces and inter process communication (IPC). Threads execute the program code within the context of an address space. Communication across address space boundaries has to be via inter-process communication or shared memory.

The Recursive Virtual Address Space Model and the three *address space modifiers*: *map*, *grant* and *unmap* are applied for construction and modification of address spaces. Thereby, *map* and *grant* are implemented as a special form of inter process communication. When a thread raises a pagefault, the fault is captured by the  $\mu$ -kernel and translated into an IPC message to the thread's pager. This pager is can resolve the page-fault by replying with a corresponding *map*-message. After that, the thread will restart at the faulting instruction.

Inter process communication is synchronous and blocking, i.e. both threads participated in the communication have to agree to the communication, the corresponding other thread blocks when this agreement is still outstanding. The L4  $\mu$ -kernel knows short and long copy messages, thereby short messages are guaranteed not to raise page faults. As mentioned above, *map* and *grant* are implemented as a special form of IPC. Instead of copying data, memory pages are transfered. As far as known to the author, the L4  $\mu$ -kernel currently achieves the best IPC performance compared to other  $\mu$ -kernels.

The L4  $\mu$ -kernel schedules threads in a prioritized timeslice-based round-robin fashion. The parameters for this scheduling can be set by user level scheduling servers. In addition to that, the L4-API supports hand-off scheduling. A thread can donate its timeslice to another thread.

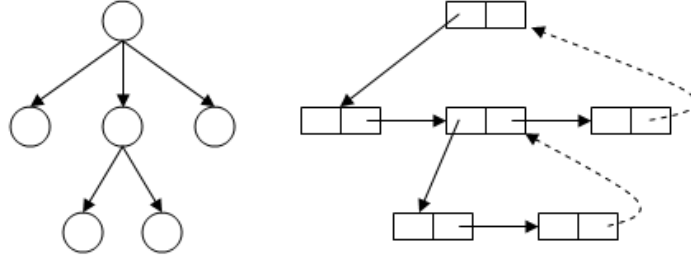
### 3.1.3 Implementations of the Recursive Virtual Address Space Model

The approach presented in this thesis relates to work presented in other implementations of the Recursive Virtual Address Space Model in the L4  $\mu$ -kernel.

As already mentioned in Section 2.6.3, common to all existing implementations is that *unmap* traverses the subtree of derived mappings in pre-order. Three different representations of the *mapping-database* tree are in use:

- *left-child, both-siblings*

This representation connects the nodes of directly derived mappings in a doubly linked list, the sibling list (see Figure 3.1). The left-child link points to the head of this list. The two sibling links at the end of the list point back to the parent

Figure 3.1: *Left child - both siblings*

node<sup>1</sup>. This data structure is in used for L4 Orangepip [Lie95], L4 Mips [Hei01] and in L4 Alpha [Uhl98].

- *pre-order doubly-linked list*

The *mapping-database* tree is sorted in a depth first search manner and the result is stored in a doubly-linked list. Each node in the list is complemented by a depth field, storing the distance of the node to the root of the tree. This data structure has been applied for L4 Hazelnut [DSU], L4 Pistachio [Tea] and for Calypso, a modified L4 Mips  $\mu$ -kernel [Szm99]. The pre-order doubly linked list is identical to the *LL* structure we propose, except for the parent-link and that *LL* is traversed post-orderly.

- *pre-order sorted array*

The Fiasco  $\mu$ -kernel [Hoh98,HH01] is a realtime capable implementation of the L4 API. Similar to pre-order doubly linked list the *mapping database* in Fiasco stores the result of a depth first search through the tree. However, instead of storing the result in a doubly linked list, it stores the mapping nodes in arrays. Differently sized arrays are allocated on demand, an insertion or deletion of nodes, and the reallocation of a different sized array, however, requires copying parts of it.

As shown in Section 2.6.3 without helping or rolling forward, a pre-order traversal of the mapping database tree on *unmap* can lead to unbounded priority inversion.

## 3.2 Synchronization

In order to ensure the consistency of the page tables and the mapping database, concurrent operations have to be synchronized.

Traditional techniques to ensure mapping database consistency are to disable the interrupts. On a uniprocessor system, this leads to executing the mapping database

---

<sup>1</sup>Tree theory calls those links “threads”. To not confuse the reader when talking about the execution entity thread, we omit the use of this term for the links.

operations non-preemptively. On a multiprocessor system, the disabled interrupts are complemented by locks protecting the mapping database.

This section surveys state-of-the-art synchronization techniques.

### 3.2.1 Overview of Synchronization Techniques

Several different algorithms and techniques have been proposed to ensure data structure consistency. Multiprocessor synchronization-techniques can be classified as follows:

**Blocking synchronization** protects critical sections with a lock. The lock prevents all other threads but the lock holder from entering the critical section. Those threads are blocked. The time, the lock is hold depends only on the lock holder and its ability to free the lock. The other, blocked threads cannot speed up this time. In particular, when preempting the lock holder, an undesirable performance degradation can be seen.

**Non-blocking synchronization** has the important property, that it does not block other threads. Non-blocking synchronization comes in two flavors: wait free and lock free.

#### Wait-Free Synchronization

Wait-free synchronization [ARJ97, Her91] can be thought of as locking, with helping replacing blocking. When a higher prioritized threads  $\tau_h$  detects, that the critical section is blocked by a lower prioritized thread  $\tau_l$ ,  $\tau_h$  helps out  $\tau_l$  by donating its priority and timeslice to  $\tau_l$  for the time it blocks the critical section.

In addition to that, wait-free synchronization guarantees the freeness of starvation<sup>2</sup>. The *wait-free locking-with-helping scheme* is one such wait-free synchronization-technique. It is used to protect the mapping database in the Fiasco  $\mu$ -kernel (see Section 2.6.5).

#### Lock-Free Synchronization

Lock-free synchronization [Val95, MS96, GC96] works completely without locks. Updates are prepared off line and atomically swapped into the lock-free data-structures. The swap fails if a conflicting update is detected. In this case, the whole operation is restarted.

The implementation of *lock-free* synchronization relies on an atomic multi-word compare and swap `mwcas` operation. This operation atomically exchanges multiple memory-words if the check succeeded. Anderson et. al [ARJ97] propose a technique to implement the `mwcas` operation in priority-based systems. Other techniques have been proposed

---

<sup>2</sup>Some authors apply the term wait-free synchronization for lock-free synchronization techniques that are starvation free. We require only freeness of starvation.

to implement `mwcas` with atomic single word compare and swap operations `cas` as they are provided by many symmetric multiprocessor systems.

Valois [Val95] investigated the lock-free implementations of the most common data-structures such as linked-lists, stacks and heaps.

### 3.2.2 Locking policies

Many locking policies have been proposed, that differ in their performance depending on the level of contention. The most common locking policies are variants of the *test-and-set-lock* [And90]. The *test-and-set-lock* shows the best performance for non-contended systems, however, its performance decreases in higher contended systems. The *test-and-test-and-set-lock* first tests whether the lock is free, before it attempts to atomically acquire it with a *test-and-set* operation.

Mellor et. al proposed the *MCS-lock* [MCS91a] to achieve a better performance in higher contended cases. Each thread atomically enqueues a list-item containing a lock variable and then spins on this lock variable locally. When releasing the lock, the lock holder dequeues its list item and hands over the lock to the next thread in the list by setting the lock variable of the next list item.

Fu et. al [Fu97] further optimized the *MCS-lock* for high contention cases by applying tree combination techniques to scale up the performance of the enqueue-operation. Threads enqueue list-items into lists at the leafs of a tree that contain only processor local lock-requests. The lists are then merged with the lists of neighboring tree nodes and propagated upwards. Once its token is in the list of the root, the lock is donated as described for the *MCS-lock*.

Lim et. al [LA94] proposes a mechanism to detect the contention level of a lock and switch between the locking policies best suited for this contention level. A high contended *test-and-set-lock* having observed several failed atomic test-and-sets, will switch for example to *MCS*.

Multi flavor locks, for example the reader-writer lock [CHP71, MCS91b] allow for parallel execution of operations of one flavor (the readers) but guarantees for mutual exclusion to the different flavors, i.e. the writers. In fact, concurrent writers are mutually excluded as well.

### 3.2.3 Uniprocessor Synchronization

Uniprocessor synchronization is simplified by the fact, that avoiding preemption during the execution of an operation makes this operation atomic.

Frequently disabling and reenabling interrupt processing is a costly operation on mod-

ern processor architectures. It pessimistically enforces an atomic execution, even if no preemption occurs. Bershad et. al [BRE92] and Druschel et. al [MDP96] propose an optimistic approach to guarantee atomic execution of operations. When a preemption event occurs, the operating system checks whether the currently executing thread is in a critical section. If this is the case, it switches back to this thread allowing it to complete its critical section atomically. After this thread completed the critical section, the operating system resumes and processes the preemption event.

### 3.2.4 Scheduler-Conscious Synchronization

Scott et. al [KWS97] proposed a combination of roll forward techniques and locking: scheduler-conscious locking. The scheduler-conscious-lock algorithm detects and reacts on preemption events. With acquiring the lock, the lock holder signals the kernel not to preempt it. Instead, it is rolled forward until it releases the lock. If in the mean time, a preemption event occurred, the lock holder detects this situation and voluntarily releases the processors by performing a *yield* operation.

## 3.3 Summary

This section introduced work, the approach presented this thesis relates to. In particular, existing implementations of the Recursive Virtual Address Space Model and synchronization techniques have been surveyed.

In the next section we present our approach, a preemptable mapping database that is free of unbounded priority inversion.

## Chapter 4

# Mapping-Database Design

This chapter presents the design of a preemptable, unbounded priority inversion free implementation of the Recursive Virtual Address Space Model. A fundamental part of this is the design of the *mapping database*, the data structure used to implement the *address space modifiers*: *map*, *grant* and *unmap*.

The important issues are:

- Choice of mapping-database representation
- Methods for concurrency control and data-structure consistency
- Preemptability
- Guaranteed Progress
- Support for multiple hardware page-sizes
- Cost of address space modifiers

Because of their complexity, *united mappings* are not discussed in this thesis. Further work is required to examine the benefits and costs of unification.

The next sections present the design for a single hardware page-size *mapping-database*. In particular it examines:

- the mapping tree representing structure,
- methods for concurrency control and to ensure data-structure consistency, and
- techniques to guarantee progress.

Multiple hardware page-size support is deferred to Section 4.4. Section 4.5 sums up the *mapping-database* design and outlines open issues.

$v$	virtual address
$\sigma_i$	address space
$PTE$	page table entry link
$D$	derived mappings

Table 4.1: *Content information stored in the mapping node.*

## 4.1 Mapping-Tree Representation

In tree and graph theory, many different representations of trees have been proposed and analyzed. This section identifies the requirements of the data structure representing the mapping-database tree and proposes three representations: *LL*, *LL-Tree* and *LL-O1*.

### 4.1.1 Mapping Node

The purpose of the mapping nodes in the mapping-database tree is to locate and modify the page-table entries of directly and indirectly derived mappings. They store the following information (see Table 4.1):

The virtual base address  $v$  of the page  $p_v$  and the address space  $\sigma_i$  are required to locate the page-table entry (PTE). Furthermore, they are required to invalidate TLB entries to make modifications to this page-table entry effective.

To avoid the page-table lookup in order to find the page-table entry, a direct pointer to the PTE can be added to the mapping node – trading space for the time required for the lookup. The above information is referred to as *content* of the mapping node.

In addition to that, the mapping node has to store *structural information*. This information is used by the *unmap* operation to iterate through the subtree of derived mappings. The *structural information* stored in the mapping node is dependent on the representation of the tree.

### Side Entry Link

When a page  $p_v$  is *mapped*, *granted* or *unmapped* from a thread in address space  $\sigma_i$ , the corresponding mapping-database node  $m(p_v, \sigma_i)$  needs to be found. This can be done by linearly searching the mapping database tree with the root node  $s_f$ . Thereby,  $s_f$  is the  $\sigma_0$ -mapping-node that corresponds to the frame  $f$ ,  $p_v$  translates to. To avoid the linear search of the mapping database, Uhlig [Uhl98] and Skoglund et. al [DSU] proposed to store direct links to the mapping nodes in the page tables.

### 4.1.2 Requirements

In Section 2.5.1 we introduced the modifications of the *address space modifiers* on the mapping-database tree.

*Map* inserts a new mapping node  $n$  and links it as a child node to the source mapping-node  $m$ .

*Grant* does not modify the mapping database tree. Instead, it modifies the *content* of the source mapping-node.

*Unmap* iterates through the subtree of derived mappings and revokes access from the pages corresponding to the mapping nodes in this subtree. If all access is revoked from a page, the page and the corresponding mapping node are removed. In the Sections 2.6 and 2.6.4, we showed that *unmap* has to iterate through the subtree of derived mappings with a post-order-traversal.

It is interesting to note, that a traversal of directly derived mappings, i.e. of the child nodes, is not required.

Because of a potentially limited stack size, the representation needs to support iterative traversal algorithms.

*Fast overmap* and *reference-information retrieval* impose additional requirements to the structure:

### Fast Overmap

Access rights can be extended to a page without implicitly *unmapping* the existing mapping if the very same mapping already exists. This is when:

1. the mapper maps from the same address space as the existing mapping,
2. the mapper maps from the same virtual source address as the existing mapping,
3. the mapper maps to the same virtual destination address as the existing mapping.

This is the case if the existing mapping of the page  $p'_v$  at the virtual destination address  $v'$  in the target address-space  $\sigma_j$  is directly derived from the mapping of page  $p_v$  at the virtual source address  $v$  in the source address-space  $\sigma_i$ , i.e. whether the mapping node  $n(p'_v, \sigma_j)$  is a child of the mapping node  $m(p_v, \sigma_i)$ . To check the above conditions, we validate this parent-child relationship.

We may search through the child nodes of  $m$  for the mapnode  $n$ . This, however, requires structural support by the tree representation to iterate through directly derived mappings.

Alternatively, we may start at the child  $n$  and look for the parent mapnode  $m$ . A *parent-link* allows to perform the *fast overmap* check directly by comparing the  $m$  to the node pointed to by  $n$ 's *parent-link*.

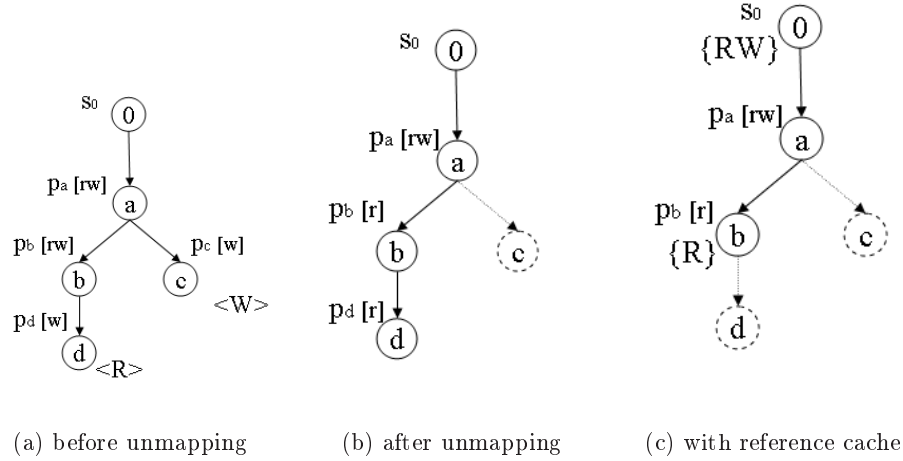


Figure 4.1: *Page-reference-information retrieval without and with reference-bit cache.*

### Reference Information Retrieval

*Unmap* returns whether the unmapped pages were referenced since they were mapped or last unmapped by a thread in the same address space as the *unmapping* thread. Therefore, the operation reads and resets the page reference information of the traversed mappings.

Figure 4.1(a) shows an example mapping database graph before and after (Figure 4.1(b)) the first unmap revoking write access from derived mappings of  $p_a$ . Square brackets show the access rights associated with the pages, the mapping nodes correspond to. Angle brackets show the page reference information. A subsequent *unmap* for example starting from node  $b$  would find the reference bits cleared, though  $d$  has been read, since it was last mapped or unmapped by a thread in  $\sigma_B$ .

As the example in Figures 4.1(a) and 4.1(b) shows, reading and resetting the reference bits is not sufficient to provide page reference information in a hierarchy of memory servers. A server that is higher up in the hierarchy resets the reference bits with *unmap* as illustrated for a memory server in  $\sigma_A$ . Underlying memory servers in  $\sigma_B$  and  $\sigma_C$  will find the reference bits reset, though the page has been accessed and modified. Even worse, memory servers can prevent servers that are higher up in the hierarchy from reading the appropriate reference information. Assume that both, a thread in  $\sigma_B$  and a thread in  $\sigma_C$  *inclusively unmaps*  $p_b$  and  $p_c$ . The result of those operations is the same as the *unmap* of  $p_a$  in Figure 4.1(b) except, that the reference information is reset and a subsequent *unmap* of  $p_a$  by a thread in  $\sigma_A$  will return that the page was not read or modified. The retrieved page reference information is wrong and therefore useless.

To solve this problem, page reference information needs to be saved for subsequent *unmap* operations before the reference bits are reset in the page tables. We propose to cache the page reference information in the mapping node. *Unmap* reads and resets the reference bits from the page tables. It logically ORs those bits with the cached page reference bits and returns the ORed reference bits read from all page-table entries traversed, including the cached values. To avoid that reference information gets lost when removing a node, the content of the cache is propagated upwards the tree. Figure 4.1(c) shows the result after the *unmap* operation revoking write access from  $p_a$ . The page reference cache of  $b$  (the content of the cache is denoted by curly brackets  $\{ \}$ ) caches the reset reference bits. The subsequent *unmap* of  $p_b$  returns that  $p_b$  has been accessed  $\{R\}$ .

The page reference cache is propagated to the parent. A parent-link allows for an efficient propagation of page reference information.

### Mapping Node Reallocation

*Mapping-node reallocation* changes the memory location, a mapping node is stored in. Usually mapping nodes are allocated once the corresponding page is mapped not reallocated until they are deallocated when the node is removed by *unmap*. In certain situations such as:

- kernel-memory management,
- kernel-resource accounting, and
- inside kernel cache management,

A reallocation of mapping nodes might be beneficial.

Reallocation of mapping nodes requires to update all inbound links that point to the mapping node. In general, reallocation is not possible for representations that derive the position of the mapping node in the mapping-database tree from the memory location the node is allocated to.

### Summary of the Requirements

To sum up the requirements, the tree representation has to support the following operations:

- **insertion** of new mapping nodes,
- **deletion** of mapping nodes,
- **post order iteration** of the subtree of derived mappings,
- **find parent**, and optionally

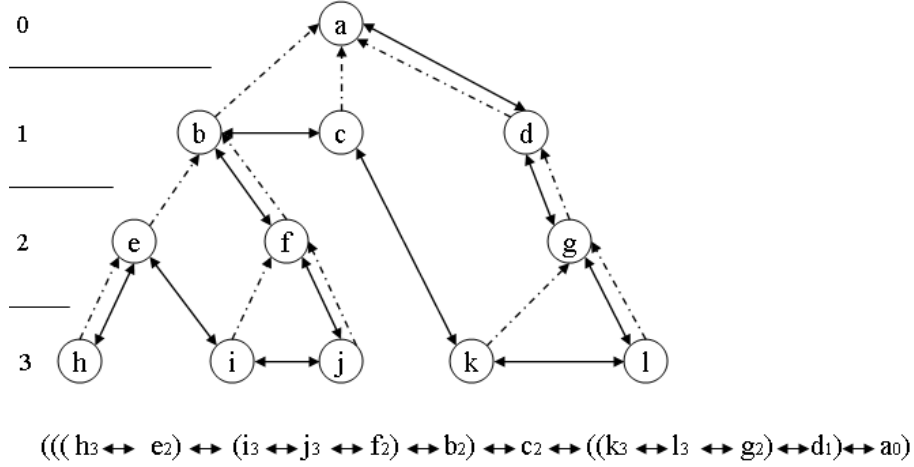


Figure 4.2: Structure of the mapping database tree representation: *LL*

- **reallocation** of mapping nodes.

The find parent operation is required for *fast overmap* and the propagation of *page reference information*. Furthermore, those operations have to be implemented iteratively.

Reallocating of mapping nodes is impossible in tree representations that derive the position of a node in the tree from the memory location it is stored in. Furthermore, compared to array-based tree representations, we expect to achieve a better performance with a pointer-based tree representation. Experimental evaluation has to substantiate this expectation.

In the following section, we will introduce three pointer-based tree representations: *LL*, *LL-Tree* and *LL-O1*.

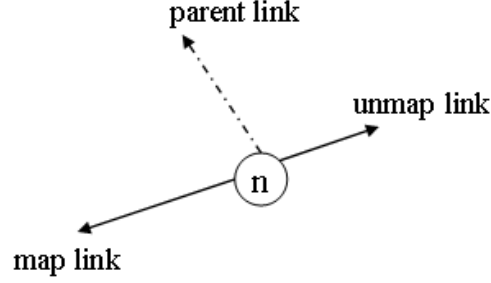
### 4.1.3 Representations

The *LL* representation is motivated by the pre-order doubly-linked list, used to represent the mapping-database tree in the L4Ka Hazelnut [DSU]  $\mu$ -kernel. *LL-Tree* and *LL-O1* are proposed to overcome the shortcomings of simple *LL*.

#### LL

*LL* stores the nodes of the mapping-database tree in a post-order sorted doubly-linked list. Figure 4.2 shows an example of a mapping-database tree stored in the *LL* representation.

*LL* has three pointers (see Figure 4.3): the *map-link*, the *unmap-link* and the *parent-link*. The *map-link* and the *unmap-link* are the opposite pointers of the doubly linked

Figure 4.3: *Pointers of LL*

list. The *parent-link* points directly to the parent mapping node, the node originates from.

In addition to those three pointers, each mapping node is complemented with its *depth*. The *depth* of a node is the distance of the mapping node to the root node of the tree. Mapping nodes that correspond to derived mappings of a node  $n$  have a *depth* that is greater than  $n$ .

Starting from the node  $n$ , the *map-link* can be followed to iterate through those derived mappings in a pre-order direction. Conversely, when starting at the left<sup>1</sup> most leaf node of the subtree with root node  $n$ , the subtree can be traversed in post-order direction by following the *unmap-link*.

The root node of the tree is the  $\sigma_0$ -mapping-node. Because  $\sigma_0$  is constructed to have an idempotent virtual to physical mapping, this  $\sigma_0$ -mapping-node directly corresponds to a frame  $f$ . The mapping nodes  $m(p_v, \sigma_i)$  of any page  $p_v$  in any address space  $\sigma_i$  that translates to the frame  $f$  are connected in this tree. The  $\sigma_0$ -mapping-node, thereby is stored at the tail of the doubly-linked list.

*Map* inserts a new node  $n$  into the tree and links it as a child to the source mapping node  $m$ . In *LL* this requires the following pointer updates in the data structure (see Figure 4.4):

The *parent-link* of  $n$  is linked to  $m$ . The *depth* of  $n$  is set to the *depth* of  $m$  plus one, i.e.  $n$  is directly derived from  $m$  and it is one level further away from the root as  $m$  is. After that,  $n$  is inserted into the doubly linked list in between the node  $m$  and the node  $o$ ,  $m$ 's *map-link* points to.

*Unmap* when revoking all access rights from a page of a mapping node  $n$  removes  $n$  from the doubly linked list of *LL* (see Figure 4.5). Therefore it links the *unmap-link* of  $o$  to the node,  $n$ 's *unmap-link* points to:  $m$ . Furthermore, it links  $m$ 's *map-link* to the node  $n$ 's *map-link* points to, i.e.  $o$ . Because *unmap* processes mapping nodes in

<sup>1</sup>“Left” means in the direction of the head of the list. In Figure 4.2 this node is drawn left from the root node of the subtree.

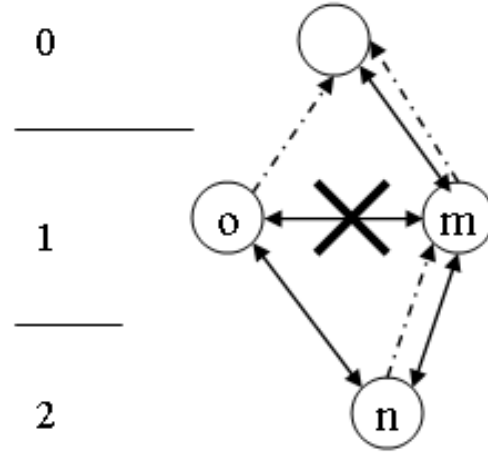


Figure 4.4: *Map* inserts the new mapping node  $n$  into the *LL* structure as a child of node  $m$ .

post-order direction, derived mappings of the node  $n$  have already been processed.

#### **find first leaf:**

Starting from the left most leaf node in the subtree to process, *unmap* can traverse the subtree to process in post-order direction by following the *unmap-link*. The problem that remains is how to find this left most leaf node, i.e. the first leaf node to start the traversal from.

The first leaf node is the left most node in the doubly linked list that has a *depth* that is greater than the *depth* of the root node. It can be found by following the *map-link* and comparing the *depth* of each node traversed to the root node.

The *find first leaf* operation has to traverse the entire subtree of derived mappings in order to find the leaf node to that the *unmap*-traversal from. Therefore, the number of nodes that have to be referenced to find the first leaf is order ( $N$ ), whereby  $N$  is the number of nodes in the subtree.

#### **fast overmap and reference-information retrieval:**

*Fast overmap* requires to validate the parent-child relationship of two mapping nodes. *Page-reference-information retrieval* requires to propagate page reference information to the reference bit cache in the parent mapping node. As already explained above, the *parent-link* of *LL* supports both operations efficiently. When choosing not to provide *fast overmapping* or *page reference information*, the *parent-link* can be omitted from the *LL* representation.

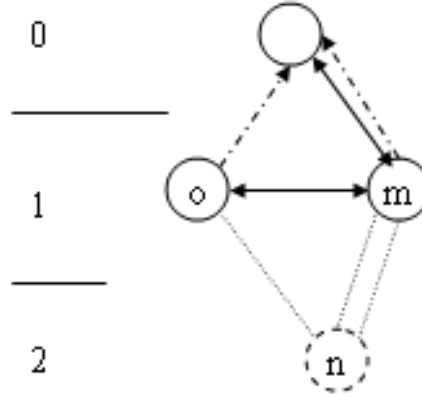


Figure 4.5: *Unmap* removes the new mapping node  $n$  from the  $LL$  structure.

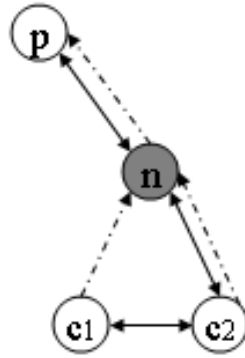


Figure 4.6: *Reallocation* of mapping nodes in the representation  $LL$

#### reallocation:

*Mapping node reallocation* does not modify the position of the node to reallocate in the tree, however, we have to update the pointers that are linked to the reallocated node  $n$  to point to the new memory location,  $n$  is reallocated to. In  $LL$  this requires the following updates (see Figure 4.6):

$n$  is linked into a doubly linked list, therefore the inbound *map-link* of the next node and the inbound *unmap-link* of the previous, i.e. left, node needs to be updates. Those nodes are directly pointed to by  $n$ 's *unmap-* and *map-link*.

In addition to that, the child nodes of  $n$  point to  $n$  with their *parent-link*. To update those, we have to iterate through all the child nodes. This is accomplished by following the *map-link* as long as the *depth* of the node traversed is greater than the *depth* of  $n$ .

Reallocation has to traverse the subtree of derived mappings in order to update inbound *parent-links*. This requires to traverse a number of nodes in the order  $(N) - N$  the number of nodes in the subtree. The update of the inbound *map-* and *unmap-link* adds two more nodes to traverse to this number. Therefore, reallocation of mapping nodes in *LL* is in the order  $(N)$ .

**space:**

*LL* needs to store three pointers in the mapping node: the *map-link*, the *unmap-link* and the *parent-link*. In addition to that, it has to store the *depth* of the node.

Compared to other pointer based structures, *LL* requires the fewest pointers to be stored and updated. However, finding the first leaf node in *LL* is order  $(N)$ .

The following representations: *LL-Tree* and *LL-O1* are modifications of *LL*. They optimize the number of nodes that have to be traversed to find the first leaf node by adding additional pointers to the data structure.

Before proceeding with *LL-Tree* and *LL-O1*, we compare the post-order traversal of operation oriented and tree oriented structures.

### Tree vs. Operation Oriented

The tree representation of the mapping database have to support post-order traversal of the subtree of nodes to process on *unmap*. *Map* and *unmap*, however, do not require to traverse the child nodes directly. This allows for two alternative representation methods: *tree-oriented representations* and *operation-oriented representations*.

A *tree-oriented* structure, we call a representation of a tree that links a node related to its parent and children. *Left child - both siblings* is an example for a *tree-oriented* structure (see Figure 3.1 in Section 3.1).

*Operation-oriented* structures not necessarily support direct links to child nodes. Instead, *operation-oriented* structures have direct links for the traversal path taken. *LL* is an example of an *operation-oriented* tree-representation.

In the two Figures 4.7 and 4.8 we compare *tree-oriented* structures with *operation-oriented* structures. Those Figures show the post order traversal path required by the *unmap* operation (dotted arrows): From the page of mapping node *a*, two direct mappings and four indirect mappings with an indirection of one level and eight with two levels of indirection have been derived. Figure 4.7 shows the post order traversal path for a *tree-oriented* representation (dashed arrows denote the additional links that have to be followed when traversing a *tree-oriented* structure). Figure 4.8 shows an *operation-oriented* representation.

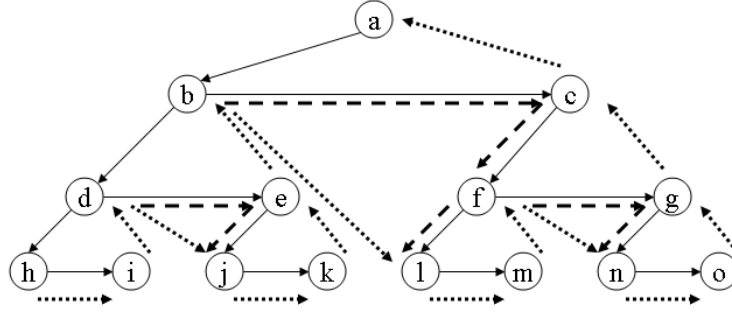


Figure 4.7: *Traversal of a tree oriented representation on unmap. Dotted arrows show the post order traversal path. Dashed show the additional links that have to be dereferenced to follow this path.*

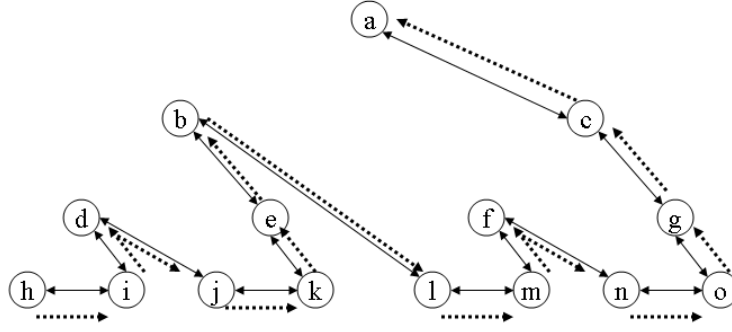


Figure 4.8: *Traversal of an operation oriented representation on unmap.*

The first leaf node this post-order traversal-path starts from is the node *h*.

The *tree-oriented* traversal algorithm removes *h*, its sibling *i* and the parent *d*. Next it finds *d*'s sibling *e* that has derived mappings. The *tree-oriented* traversal algorithm has to traverse down this subtree to find the leaf *j*, then *k* and finally *e*. Completed with *b*'s subtree, *b* can be removed but progressing with *c* requires to traverse down to *f* and *l* before being able to remove *l*, *m*, and *f*. A last time, the *tree-oriented* algorithm has to search for *n* before removing *n*, *o*, *g*, and finally *c* (and *a* if it was a *inclusive unmap* operation).

An *operation-oriented* traversal algorithm omits the intermediate searches for leaf nodes. Instead the traversal path for *unmap* is explicitly given (see Figure ??). Given the first leaf *h*, the operation oriented algorithm is able to directly revoke the subtree of *a*: *h*, *i*, *d*, *j*, *k*, *e*, *b*, *l*, *m*, *f*, *n*, *o*, *g*, *c*, and finally *a* on an *inclusive unmap*.

To summarize, post-order traversal in *tree-oriented* representations need to traverse to

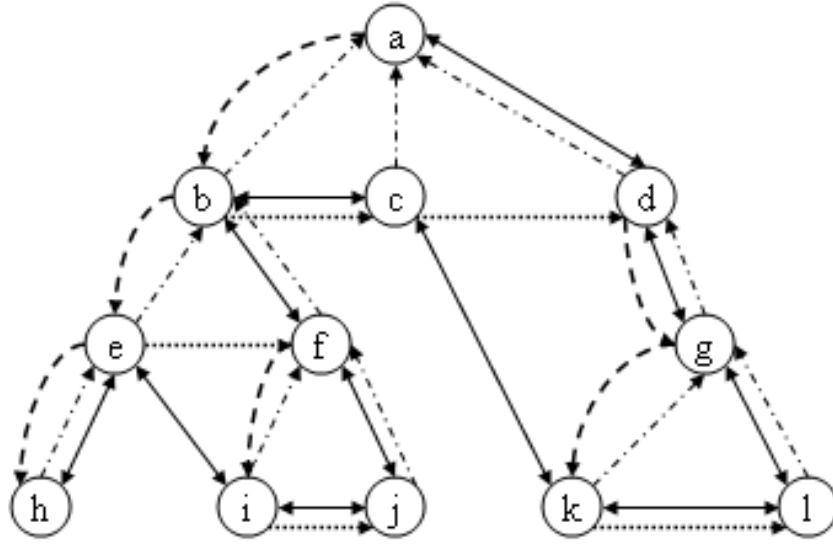


Figure 4.9: Structure of the mapping database tree representation: *LL-Tree*

the first leaf of any subtree in the subtree of nodes that are processed by *unmap*. In an *operation-oriented* representation, only the first leaf has to be located.

Because of this, *operation-oriented* representations like *LL* are preferable, assuming the first leaf node can be located sufficiently fast. In *LL* this requires to traverse order  $(N)$  nodes, whereby  $N$  is the number of nodes in the subtree to be processed by *unmap*. *LL-Tree* and *LL-O1* optimize the operation to find the first leaf.

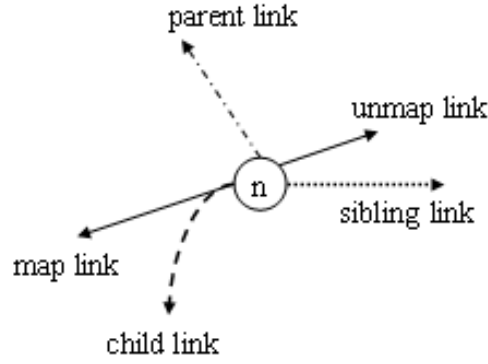
### LL-Tree

*LL-Tree* extends the tree representation *LL* with two additional pointers to speed up the search for the first leaf node (see Figure 4.10: a (left-) *child-link* and a *sibling-link*).

The *child-link* of node  $n$  points to the left most child node that originates from  $n$ . All child nodes of a node  $n$  are connected in a singly linked list. The *sibling-link* connects those nodes in the list. Thereby the *sibling-link* points to the next sibling on the right. The *child-link* points to the head of this list.

#### find first leaf:

Instead of having to traverse through the entire subtree of derived mappings, *LL-Tree* can find the first leaf node by following the *child-links* until the *child-link* of a node  $n$  points to no further node, i.e. is *null*. This node  $n$  is the first leaf node to start the post-order traversal from. The *child-link* makes it obsolete to store the *depth* in the mapping node.

Figure 4.10: *Pointers of LL-Tree*

The number of nodes that have to be traversed in *LL-Tree* to find the first leaf node is in the order of the depth of the first leaf node relative to the root node of the subtree to traverse. With each *child-link* followed, the find first leaf operation covers one level of mapping nodes.

The possibility to find the first leaf node faster comes at the costs of updating the additional pointers. Insertion of nodes on *map* and removal of nodes on *unmap* have to update the *child-* and *sibling-link* as follows:

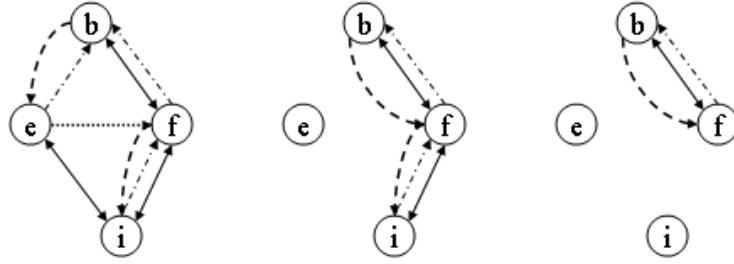
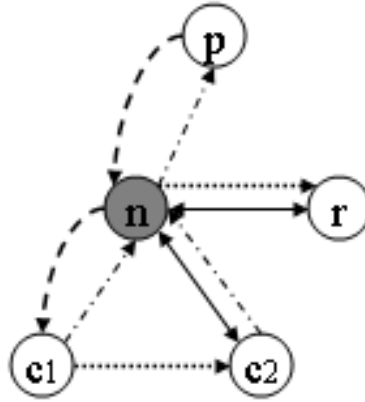
**insert node:**

A *map* operation starting from node  $m$ , inserts a new node  $n$ . In this case we have two situations:  $m$  has no child nodes mapped, or  $m$  already has a child node mapped. In the first case, the *child-link* of  $m$  is set to point to  $n$ . In the second case, the *sibling-link* of the right most child of  $m$  has to be updated to point to  $n$ . In this case, prior to inserting  $n$ , the *map-link* of  $m$  points to this right most sibling-node.

**remove node:**

When removing the left most child  $n$  of a mapping node  $m$ , the *child-link* of  $m$  needs to be updated. After having removed  $n$ , the new left most child node of  $m$  is the right sibling of  $n$  (see Figure 4.11). Therefore, we set  $m$ 's *child-link* to point to the node,  $n$ 's *sibling-link* points to.

When removing another child node  $o$  of  $m$ , we do not have to update the *child-link*. Instead, the *sibling-link* of the left sibling needs to be updated. Post order traversal on *unmap* guarantees, that all derived mappings of  $o$  already have been removed. In this case  $o$ 's *map-link* points to the left sibling  $l$ . We update  $l$ 's *sibling-link* to point to the

Figure 4.11: Update of the child-link when removing a node from *LL-Tree*Figure 4.12: Update of the child-link on mapping node reallocation in *LL-Tree*

node,  $o$ 's *sibling-link* points to.

#### reallocation:

In addition to update the inbound *map-* and *unmap-links*, *parent-links* – the updates of those pointers has already been described for *LL* – a reallocation of a mapping node  $n$  in *LL-Tree* has to update inbound *child-* and *sibling-links*. Note, reallocation has to update either the inbound *child-link* or the inbound *sibling-link*, but not both.

If the node  $n$  to reallocate is the left most child, the inbound *child-link* of  $n$ 's parent  $p$  needs to be updated (see Figure 4.12).

If  $n$  is not the left most child, the *sibling-link* of the left sibling needs to be updated (see Figure 4.13). This left sibling  $l$  has to be found by searching for the first leaf  $c_1$  and dereferencing the *map-link* of  $c_1$ .

*LL-Tree* allows a faster update of the inbound *parent-links* by iterating through the linked list of siblings,  $c_1$  and  $c_2$ . The head of this list is the node,  $n$ 's *child-link* points to.

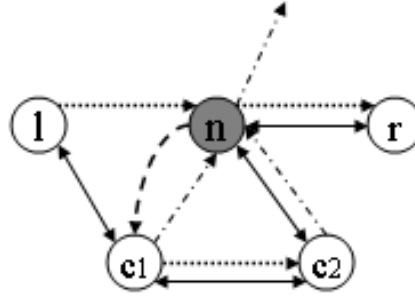


Figure 4.13: Update of the sibling-link for mapping node reallocation in *LL-Tree*

Reallocation of a mapping node  $n$  in *LL-Tree* requires to traverse order  $(C+D)$  nodes, whereby  $C$  is the number of child nodes of which the *parent-link* have to be updated.  $D$  is the depth of the first leaf node relative to the root of the subtree  $n$ . The traversal of those order  $(D)$  nodes is required only if the *sibling-link* has to be updated.

**space:**

*LL-Tree* needs to store five pointers in the mapping node: the *map-link*, the *unmap-link*, the *parent-link*, the *child-link* and the *sibling-link*.

Compared to *LL*, *LL-Tree* may find the first leaf node faster. However, two additional pointers have to be updated. *LL-Tree* trades the space required to store the additional pointers and the time to update them against the time to search for the first leaf node.

Can we do better in finding the first leaf than order:  $O(\text{subtree depth})$  ?

**LL-O1**

*LL-O1* is based on the following observation (see Figure 4.14):

When searching the left most leaf node of a subtree by traversing a first-child list as it is done in *LL-Tree*, all subtrees with the root-node in the list ( $a$ ,  $b$ ,  $d$ , and  $f$ ) share the same first leaf node  $f$ . The key idea of *LL-O1* is to provide a direct link to this first leaf node: the *down-link*. This link is stored in the node at the head of this first-child list:  $a$ . *Up-links* of the other nodes point to this head.

*LL-O1* has the following pointers:

In addition to the *map-*, *unmap-*, *parent-* and *sibling-link* of *LL-Tree*, *LL-O1* introduces the *down-link* and the *up-link* (see Figures 4.15 and 4.16). The *child-link* of *LL-Tree* is not required.

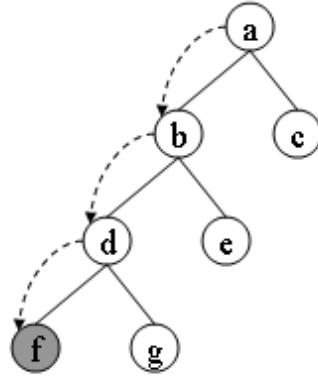


Figure 4.14: The left most leaf-node to start unmap traversal from is shared by nodes in the first child list.

#### find first leaf:

In the common case, the first leaf node of the subtree of derived mappings with root node  $n$  is pointed to by the *down-link* of the node,  $n$ 's *up-link* points to. In this case, the first leaf is found in order (1).

Certain difficulties when removing nodes (see below) might require to traverse a list of *down-links*. In the worst case, this results in having to traverse order ( $D$ ) nodes, whereby  $D$  is the depth of the first leaf relative to the root  $n$  of the subtree to process on *unmap*.

#### insert node:

When inserting a new node  $n$  as a child of a node  $m$ , *up-* and potentially the *down-links* needs to be updated. There are three potential cases:  $m$  already has a child node mapped,  $m$  has no child mapped and is a left most child node, or  $m$  has no child mapped but is not a left most child node.

In the first case no updates are required.

In the second case,  $m$ 's *up-link* points to a node  $o$  that stores the shared *down-link*. In this case,  $n$ 's *up-link* is set to point to this node  $o$  and  $o$ 's *down-link* is updated to point to the new first leaf  $n$ .

In the thirist case,  $m$  would be the head of the left most child list in *LL-Tree*. In this case, we set  $m$ 's *down-link* to point to  $n$  and  $n$ 's *up-link* to point to  $m$ .

#### remove node:

When removing the first child node  $n$ , the *down-link* that points to  $n$  is updated to point to  $n$ 's sibling node if  $n$  has a sibling and otherwise to  $n$ 's parent node.

*Inclusive unmap* operations, preempted *unmap* operations, and *unmap* operations that revoke only a subset of the access rights mapped, might lead to the following anomalies.

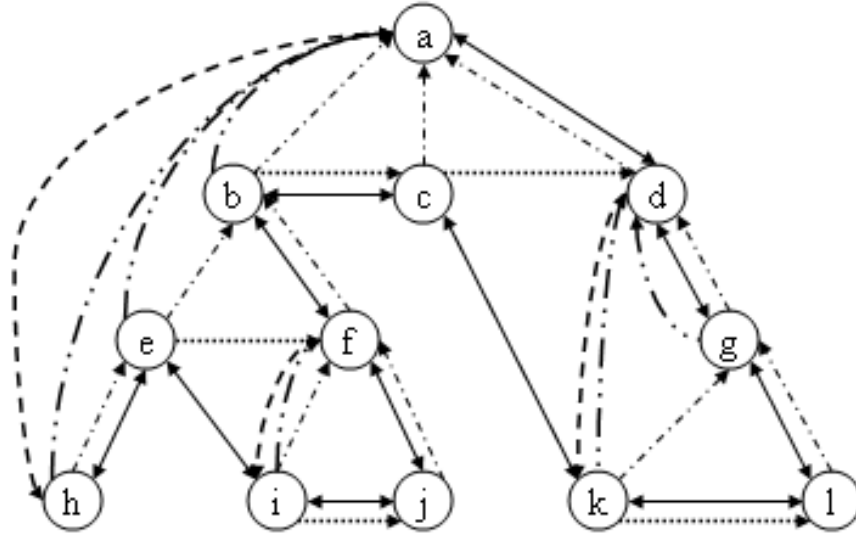


Figure 4.15: *Structure of the mapping database tree representation: LL-O1*

The following example elucidates this effect (see Figure 4.17):

An *inclusive unmap* of mapping node  $b$  results in node  $d$  becoming the common first leaf of the subtrees with root nodes  $a$ ,  $c$  and  $d$ . An *inclusive unmap* of  $b$ , however, updates  $a$ 's *down-link* to point to  $c$ , and  $c$ 's *up-link* to point to  $a$ .  $c$ 's *up-link*, however, still points to  $c$  instead to  $a$ . Therefore  $c$ 's *down-link* has to reference  $d$  instead to be *null*.

A similar effect results from an *unmap* operation that manages to remove  $b$ , but is preempted before being able to remove  $d$ .

This anomaly effects the performance of finding the first leaf, as shown above.

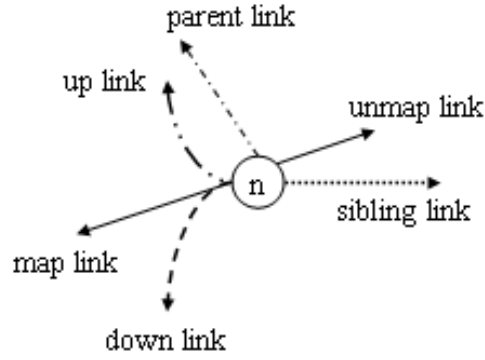
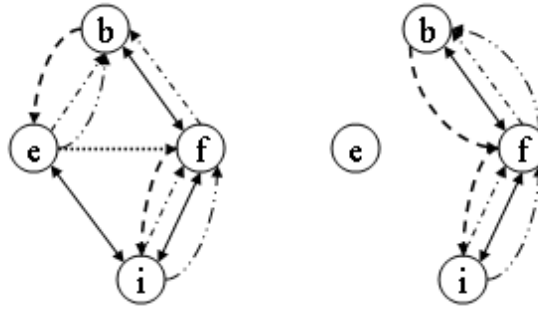
#### reallocation:

The difficulty in reallocating a mapping node  $n$  in *LL-O1* is to update inbound *up-links*. This is the case if  $n$ 's *down-link* is not *null*. The inbound *up-links* are updated by dereferencing  $n$ 's *down-link* and traversing through a list that is stretched by the *parent-link*. The *up-link* of each node in the list is updated. The list is traversed, until  $n$  is reached.

The reallocation costs are the same as for *LL-Tree*, i.e. order  $(D + C)$ , whereby  $D$  is the depth of the first leaf relative to the root of the subtree,  $C$  is the number of children of the node to reallocate.

#### space:

*LL-O1* is the largest of the three data structures. It requires to store 6 pointers in the

Figure 4.16: *Pointers of LL-O1*Figure 4.17: *Flush and preempted unmap lead to LL-O1 degrade to LL-Tree.*

mapping node.

Compared to the other two representations *LL* and *LL-Tree*, in the best case *LL-O1* requires to traverse only order (1) nodes to find the first leaf node. In the worst case it requires to traverse order ( $D$ ) nodes, whereby  $D$  is the depth of the first leaf relative to the root of the subtree. This however, comes at the cost of storing and updating 6 pointers.

Like *LL-Tree*, *LL-O1* trades the space required to store the additional pointers and the time to update them against the time to search for the first leaf node.

#### 4.1.4 Summary Tree Representation

Three operation-oriented presentations have been introduced for the mapping-database: *LL*, *LL-Tree* and *LL-O1*. *LL-Tree* and *LL-O1* trade space and time to update additional pointers for a better performance to find the first leaf node of the subtree processed

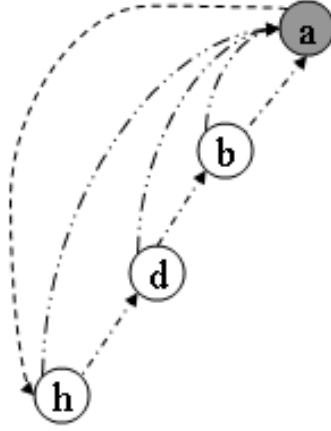


Figure 4.18: Update of inbound up-links for mapping node reallocation in LL-O1

representation	number of pointers	references to find the first leaf	costs of reallocation
<i>LL</i>	3 plus depth	$O(N)$	$O(N)$
<i>LL-Tree</i>	5	$O(D)$	$O(D + C)$
<i>LL-O1</i>	6	$O(1)$ best case, $O(D)$ worst case	$O(D + C)$

Table 4.2: Summary of the properties of the three proposed mapping-database representations. Thereby,  $N$  is the number of nodes in the tree,  $D$  is the depth of the first leaf node relative to the root of the subtree,  $C$  is the number of children of the node to reallocate.

by *unmap*. Table 4.2 summarizes the properties of those structures. Note, that only pointers representing the tree structure are listed. In addition to that, the following payload information listed in Table 4.3.

Open issues concerning the tree representations are:

- Cost of modifications by *map*, *grant* and *unmap*
- Multiple hardware page size support

## 4.2 Concurrency and Consistency

To ensure the consistency of the *mapping database* and the page tables, conflicting operations on the *mapping database* have to be synchronized. Several different algorithms and techniques have been proposed to ensure data structure consistency. Those can be classified into *blocking* and *non-blocking* synchronization techniques (see also

$v$	virtual address
$\sigma_i$	address space
$pte$	page table entry link
$RWX$	reference cache

Table 4.3: *Payload information in the Mapping Node.*

Section 3.2). *Non-blocking* synchronization primitives can further be classified into *wait-free* and *lock-free* techniques.

Special solutions have been proposed for uniprocessor systems:

- *disabled interrupts*, and
- *roll forward techniques*.

The relevant questions to control concurrent operations on the *mapping database* and thereby ensure data-structure consistency are:

- the granularity of synchronization, and
- the choice of the appropriate synchronization technique.

#### 4.2.1 Granularity

The modifications of the mapping database and the page tables, when *mapping* and *granting* of a page and *revoking access* from the page corresponding to a single mapping node, have to be performed as atomic operations.

*Unmap* is preemptable after processing a single mapping node.

It is possible to weaken this atomicity requirement if a concurrent operation is able to take over and complete the preempted operation to bring the *mapping database* into a consistent state.

Locking in the *mapping database* can be done in three different granularities:

- the entire database,
- the frame, or
- the mappings operated on.

#### Mapping Database Lock

The *mapping-database lock* is the most coarse-grain lock. Only a single thread is allowed to perform *mapping-database* operations. Concurrent *mapping-database* operations are prevented from operating on the database, until the lock is released.

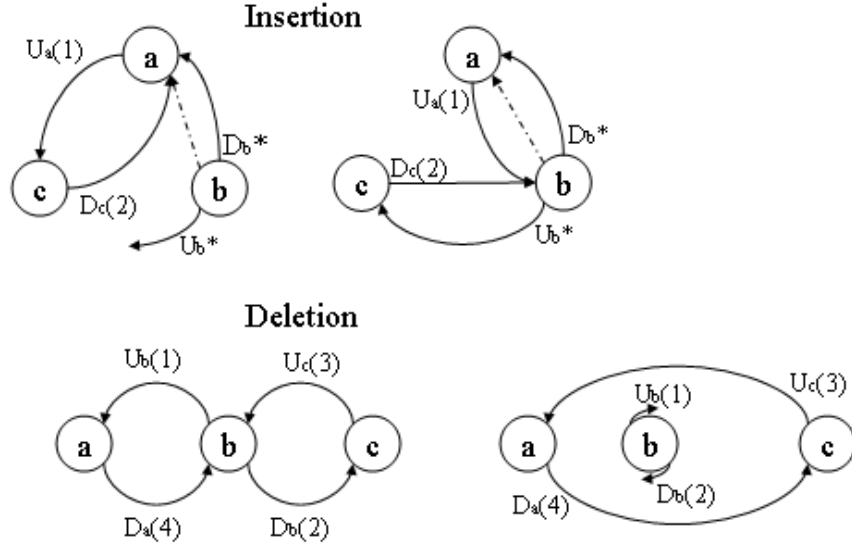


Figure 4.19: Two phase locking protocol for insertion and deletion in LL

### Frame Lock

*Frame-locks* add a lock to each page-frame. The *frame-lock* protects the *mapping-database* tree with the  $\sigma_0$ -root-node, that corresponds to the frame. Concurrent operations *mapping*, *granting* or *unmapping* a page  $p_v$  that translates to the locked frame  $f$ , are prevented from execution, until the frame lock of  $f$  is released. *Frame-locks* are medium-grained locks.

### Node Lock

*Mapping-node locks* are the most fine-grained locks. A per node lock protects this node, its pointers and the corresponding page-table entry.

*Map* and *unmap*, however, have to lock multiple nodes to atomically insert or remove nodes. Skoglund et. al [DSU] (see also the “ $\mu$ -kernel construction” lecture notes [Uni00]) proposed a two phase locking protocol to avoid deadlocks during the insertion and deletion of mapping nodes:

Figure 4.19 shows insertion and deletion and the order the locks are acquired in.

Each mapping node is protected by two locks: an *up-lock* and a *down-lock*.

Applied to *LL*, the *up-lock* protects the mapping node, the *map-link*, the *parent-link* and the corresponding page-table entry. The *down-lock* protects the *unmap-link*. *Up-locks* can be held. *Down-locks*, however, have to be released if the thread does not manage

to acquire the corresponding *up-lock*.

A coarse-grained mapping-database lock prevents all but one operation from execution. In a symmetric multiprocessor system, this would result in a serialized processing order of memory-management requests. Even though, those requests operate on different parts of the mapping database, not conflicting each other, a parallel execution is prevented.

Frame granular locking requires for those operations to acquire a single lock as well, the lock corresponding to the frame. However, frame locks allow for a parallel execution of operations that process another frame.

Compared to node-locks, a frame-lock trades the possibility to allow for several operations to process the frame in parallel against the performance increase of having only to acquire a single lock for the operation on the page.

As long as the level of contention on the frame-lock allows for acquiring the lock sufficiently fast, frame-locks are to be preferred. We expect a low level of contention on the frame-locks. Section 4.2.2 below presents a roadmap of possible alternatives if this expectation would not hold.

## 4.2.2 Synchronization Techniques

This section discusses the choice for the appropriate synchronization method for frame-granular synchronization.

### Blocking

Blocking prevents other operations from processing for the time the critical section lasts. When threads are blocked even by preempted threads, blocking may lead to unbounded priority inversion as the following example shows:

Assume a lower prioritized thread  $\tau_l$  acquires a lock and is preempted. A higher prioritized thread  $\tau_h$  attempts to acquire the same lock, but finds it blocked by  $\tau_l$ . Priority is inverted. The priority inversion is unbound, because we do not require the scheduler of our system to reschedule  $\tau_l$  in a limited time again.  $\tau_l$  is not able to release its lock.

### Wait Free vs. Lock Free

Lock free synchronization extensively relies on an atomic multi word compare and swap (MWCAS) operation. Techniques have been proposed to implement MWCAS on top of single or double word compare and swap, however, those techniques have a considerable overhead in space and time [ARJ97].

Wait-free synchronization, however, requires to “help” the lower priority lock-holders out of their critical sections in order to not block. As discussed in Section 2.6.5, helping leads to undesirable system behavior and requires complex implementations – especially

on multiprocessor systems.

We propose a combination of scheduler-conscious synchronization [KWS97] with roll forward techniques [BRE92, MDP96] on the individual processors. Operations are rolled forward and all locks are released prior to preempting an operation. Preempted operations cannot block other operations. Because the operation is completed non-preemptively and releases all locks prior to a potential preemption, no helping schemes are required to make the lock-holder complete its critical section.

### Locking Policy

With the assumption of low contention per frame this section describes the locking policy applied to the *mapping database*. It is a combination of roll-forward techniques with scheduler-, i.e. preemption-conscious locking.

The *address space modifiers*: *map*, *grant* and *unmap* are implemented inside the  $\mu$ -kernel. Therefore, no trust issues arise as they do when applying roll-forward techniques for user-level applications. The  $\mu$ -kernel has to be trusted anyway, not to monopolize the system. Therefore, the issue of limiting the allowed roll-forward time does not come up.

The issue of detecting when to roll forward remains. Several techniques have been proposed to detect the regions of code that needs to be rolled forward. For the reason of simplicity, we signal the need to roll forward by setting a flag.

When starting with the code that is to be rolled forward, the threads sets a flag: the “unpreemptable-flag”. Upon an preemption event, the processor checks for this flag being set and resumes the threads operation if this is the case. At the end of the critical section, the thread resets the “unpreemptable-flag”. If a preemption occurred, the thread voluntarily releases the CPU with a *yield* operation (`thread_switch (NilThread)` systemcall) to allow for the pending preemption to occur. The situation, that a preemption occurred while operating the critical section is signaled to the thread by a second flag: the preemption pending flag.

The disadvantage of this simple technique is that additional instructions are required to set the flag when entering the roll-forward path. A more optimistic approach would require overhead to detect the roll forward situation first after a preemption happened. When assuming infrequent preemptions this solution is preferable.

For uniprocessor systems, the roll-forward technique described above is sufficient for synchronization. Multiprocessor systems require to further means of synchronization: *frame-locks*.

We apply a scheduler-conscious frame-granular locking algorithm for multiprocessor synchronization in the mapping database. We lock the frame that corresponds to the mapping database tree operated on for the time it takes to process a single mapping

node and the corresponding page tables. The entire insertion and modification of the mapping nodes and the corresponding page tables of a single page *map* and *grant* is rolled forward and protected by the frame lock. Preemption is made possible, with all locks released after each single page operation is completed. *Unmap*, processing multiple mapping nodes even for *unmapping* a single page, is preemptable after having processed a single mapping node. The frame lock is released at the end of processing the single node and it is reacquired for the next node to flush.

The property of scheduler-conscious synchronization techniques [KWS97] is, that by the time the lock is acquired, the lock holder is no longer preemptable. The “unpreemptable-flag” is set. Prior to clearing this flag, the *frame-lock* is released.

To avoid starvation, a starvation-free locking algorithm has to be selected, for example the scheduler-conscious ticket lock.

### High Contention: a Roadmap

The assumption of having low contention on a frame has lead to frame-granular locking-scheme for the *mapping database*. This section discusses options that remain if the level of contention increases.

If the contention level increases, we assume that it varies over time and that threads appear different contention levels dependent on the frame or mapping node they are operating on. The reasoning for this assumption is based on the different usage schemes of the page frames. A page frame used to store the code of an application in, for example is low contended. However, when the application terminates or is swapped out and the page frame is reused as a buffer for a server that frequently *maps* and *unmaps* it to and from its clients, contention may increase. The same applies for nodes in a hierarchical system. The node of a lower level swapping server might for example be low contended, the node representing the mapping in the server described above, however, might be higher contended.

We see the following solutions to improve mapping database performance on higher contention:

**Reactive locks** [LA94] switch between different locking algorithms that are better suited for the level of contention currently appeared. Locking a low contended frame would use a locking algorithm that can fast and efficiently lock the node (for example the test-and-test-and-set lock), while acquiring a high contended lock would apply an MCS-lock algorithm [MCS91a].

**Node-granular locking** allows to reduce contention on the frame, since operations on independent nodes can be executed in parallel. The tradeoffs in changing the

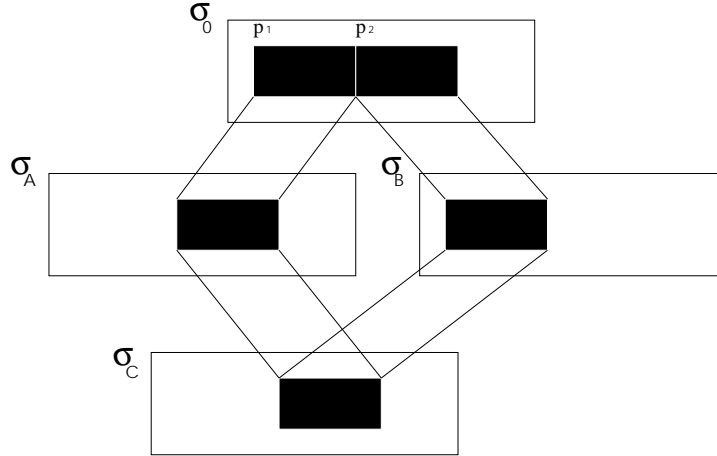


Figure 4.20: The page table entry in  $\sigma_C$  needs to be protected by a page table entry lock.

locking granularity to node-granular locking are the costs to acquire the several locks needed.

**Multi-flavor-locks** like the reader-writer-lock, mutually exclude operations of different flavors. Operations of the same flavor as the lock-holders might be tolerated (for example multiple readers) but does not need to (only a single writer is allowed).

For mapping-database synchronization we can apply a similar multi-flavor-locking-scheme. Concurrent *map* operations insert new nodes to a target node  $t$ . To insert the new node, *map* needs to update the *map-link* of  $t$  and the *unmap-link* of the left node, i.e. the node  $t$ 's *map-link* points to. Parallel insertions on all nodes but  $t$  operate on different *map*- and *unmap-links*. Therefore, parallel *maps* can be allowed when protecting the mapping nodes (here  $t$ ) with an additional fine-grained lock. The purpose of this lock is to prevent parallel *map* operations of the same page by threads in the same address space. *Unmap* is executed mutually excluded from *map* operations and from other *unmap* operations.

### 4.2.3 PTE Lock

With *frame*- as well as with *node-locks* a single special case remains that requires to lock the page table entry (PTE) of the target mapping with a *PTE-lock* (see Figure 4.20). Assume two map operations (by threads  $\tau_1$  and  $\tau_2$ ) concurrently map two different frames to the same page in the target address space. This situation may arise if two threads in the receiving address space accept mappings to the same location. Further, assume a conflicting mapping already exists in the target area, we have to *overmap*.  $\tau_1$  and  $\tau_2$  both manage to acquire the *frame-lock* and  $\tau_1$  finds and *inclusively*

*unmaps* the target entry.  $\tau_2$  now checks its target page-table entry and finds the region free because  $\tau_1$  completed the flush.  $\tau_2$  maps its page and returns.  $\tau_1$ , however, since it completed its flush assumes a free target area as well, which is not true, because of the mapping of  $\tau_2$ . This is because the target area is not protected by neither of the two thread's *frame-locks*. We have to protect this target area by an additional *PTE-lock*

### 4.3 Preemptability and Progress

Preempted *address space modifiers* have to leave the *mapping database* in a consistent state. *Map* and *grant* are preemptable after having processed a single page. Those operations are easy to restart. The operation on the previous page was completed before the operation was preempted. When rescheduling the thread performing those operations, the *map* or *grant* operation can resume processing the next page. Restarting *unmap*, however, is more complex. This section deals with the restart of a preempted *unmap* operation. The restart mechanism has to fulfill the constraint to guarantee forward progress in order to guarantee freedom of starvation.

#### 4.3.1 Restart of Unmap: Problem Analysis

The simplest solution is to restart the preempted *unmap* operation from the beginning. The mapping tree is consistent, so *unmap* can traverse through the subtree and revoke access from the pages corresponding to the nodes in it. When revoking partial access, nodes can be found where access was already removed from. However, there is no difference whether those nodes have been mapped with weaker rights, or a previous *unmap* already revoked the rights. The problem with this solution is, that *unmap* can starve if subsequent preemptions prevent *unmap* from ever completing its traversal through the subtree.

By guaranteeing forward progress we can guarantee starvation-freeness.

Since we cannot guarantee forward progress when restarting the *unmap* operation all over, *unmap* needs to be restarted at the node within the tree that it has to process next. Unfortunately, a concurrent *map* or *unmap* operation might change the node to restart from. The following two scenarios may arise:

- A *map* operation inserts a new child node  $e$  to the node  $b$  to restart from. Because of the node  $b$  to restart from has not been processed yet, this child node  $e$  might be mapped with an access right that the preempted *unmap* operation revokes. Therefore, we have to restart the *unmap* operation at node  $e$  instead of  $b$ .
- A concurrent *unmap* operation may remove the mapping node  $b$ . In this case the

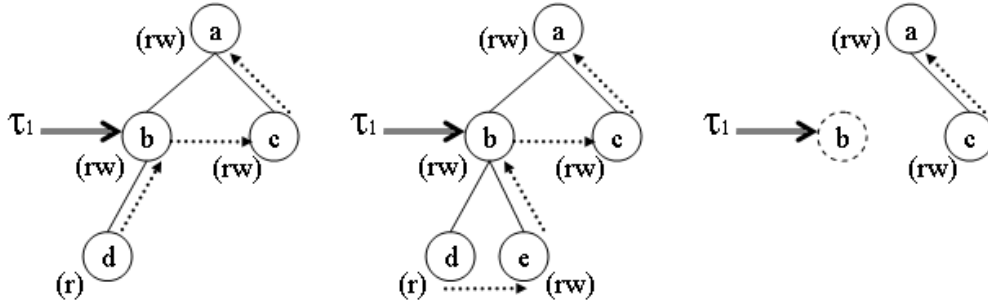


Figure 4.21: The restart point of a preempted unmap has to change if new nodes are inserted or if the node to restart at is removed.

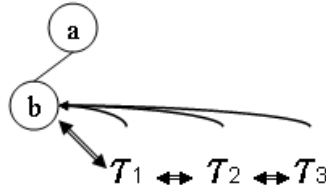


Figure 4.22: The preempted-thread list links the TCBs of preempted unmap operations in a doubly linked list and to the mapping node they have to restart from. Each TCB contains a direct link to the node to restart with.

restart-point of the preempted *unmap* has to be updated to the node to process next, i.e. to *e*.

The key idea of the following solutions is to publish and update the restart point of preempted operations on *map* and *unmap*. We call this operation *restart-point tracking*.

The issues in *restart-point tracking* are:

- to find the preempted threads whose preemption points need to be updated, and
- the costs of updating the preemption points.

We propose the following two solutions:

- the *preempted-thread list*, and
- the *token-based preempted-thread list*.

### 4.3.2 Preempted-Thread List

Before being preempted, the unmapping thread inserts its thread control block (TCB) into a doubly-linked list, the *preempted-thread list* of the node to restart from. In ad-

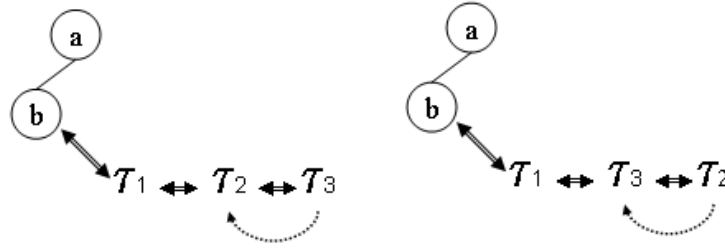


Figure 4.23: *Omitting the direct link to the mapping node to restart with can lead to starvation.*

dition to that it sets a link in the TCB to point to this node (see Figure 4.22). An additional link in the mapping node point to the head of this list.

A *map* operation inserting a new node  $n$  or a *unmap* operation prior to removing the node walks this list and updates the restart-point. Therefore, the operation traverses through the list and updates the link in each TCB to point to the new node to restart from, i.e.  $n$  for *map* and the node  $m$  that is to be processed next for *unmap*. In addition to that, the *preempted-thread list* is merged with the *preempted-thread list* of the new restart point.

The costs to update the preemption point are in the order of the number of preempted threads in the list. Note, that those costs add to the performance of *map* and *unmap*. In the worst case, this can be up to order of the number of threads in the system, when all threads but one are preempted on the same mapping node.

The restart link in the TCB and the fact, that a TCB is linked to the *preempted-thread list* are redundant information. The linked list is required to identify the threads to update. Omitting the TCB link leads to the following algorithm:

The *mapping node* points to the head of a linked-list of TCBs. Updating the preemption point then requires to only merge the list with the list at the new restart point. When restarting a preempted *unmap*, the preempted thread needs to search for the head of the list to determine the restart point. This solution trades restart performance for update performance.

With this simple thread list, however, we cannot guarantee to avoid starvation while allowing for the search of the restart point itself to be preemptable.

Figure 4.23 illustrates this in the following example: Assume three preempted threads  $\tau_1$ ,  $\tau_2$  and  $\tau_3$  in the simple thread list.  $\tau_3$  is rescheduled and manages to move one step forward in the list, i.e. it is in between  $\tau_1$  and  $\tau_2$ , before it is preempted again.  $\tau_2$  is woken up and does the same as  $\tau_3$ . Leading to an identical situation as in the beginning.

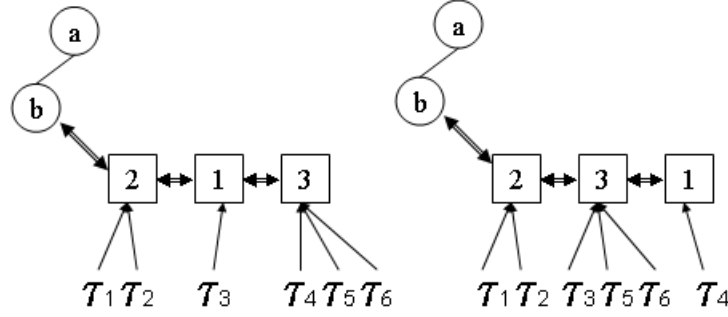


Figure 4.24: An indirection to shared tokens allows to omit the direct link to the mapping node in the token based preempted thread list.

If this situation persists for subsequent activations of  $\tau_2$  and  $\tau_3$ , both threads starve.

We propose the *token-based preempted-thread list* to overcome both the starvation problem and the list traversal of the preempted-thread list.

### 4.3.3 Token-Based Preempted-Thread List

The *token-based preempted-thread list* adds one level of indirection between the TCB and the mapping node (see Figure 4.24).

The link to the restart point is omitted from the TCB. Instead, a link to a token is added. Instead of the TCBs, those tokens are linked into a cyclic doubly-linked list. The head of this list is furthermore doubly linked to the mapping node to restart from. The fundamental difference to linking the TCBs is, that tokens can be shared.

A reference count at each token denotes the number of threads sharing this token. When a preempted *unmap* operation is rescheduled, the thread updates the token link in the TCB to point to the next token. It decreases the reference count of the old token and increasing the reference count of the next token. Tokens are removed from the list and deallocated when they are no longer shared, i.e. the reference count becomes zero.

By rolling forward this operation, the restarted *unmap* is guaranteed to progress one step further to the restart point. Since the tokens are shared other operations cannot lead to a situation like described for the simple thread-list. Once the restarted *unmap* reaches the token at the head of the list, it continues with its access revocation.

*Map* and *unmap* update the restart points by merging the *token-based preempted-thread list* with the *token-based preempted-thread list* of the new restart point. This can be accomplished by updating only a few links. Therefore, *preemption-point tracking* can be accomplished in order one.

Restarting the preempted *unmap* requires to traverse through the list which takes in the worst case a number of steps in the order of the number of threads in the system. This operation is preemptable after each step, i.e. when the TCB link has been moved one token further. In the common case, a single token shared by the preempted TCBs is expected.

### Root Node Overrun

The restart point of preempted *unmap* operations is tracked and updated. A concurrent *unmap* operation might, have removed the entire subtree the preempted *unmap* has to process. Even worse, in the mean time, a completely different subtree might have been established. This situation is called *root-node overrun*. It can be characterized by the fact, that the *root mapping-node* of this subtree has been removed. *Preemption-point tracking* still updates the restart points of this preempted *unmap* operation, because otherwise it would have to check for the *root-node overrun*, i.e. to traverse the list. Instead, an independent detection mechanism is required for *root-node overrun* detection.

The key idea to detect the overrun, is to signal the condition to the *unmapping* operations with a flag. This is accomplished by adding a reference count and a root-overrun flag to the mapping node. The *unmap* operation, removing the mapping node checks whether the node removed is a root mapnode of a preempted operation. In this case, the reference count is greater than zero. If so, it sets the root-overrun flag to denote the overrun instead of deallocating it. The node is completely removed from the mapping-database tree by the *unmap* operation. Other *unmap* operation will therefore find a consistent tree. However, the memory space of the node is not yet released.

Preempted *unmap* operations, when being rescheduled, check their root mapping node for the flag set and if so, decrease the reference count and return. The last thread, i.e. the thread decreasing the reference count to zero, deallocates it.

The kernel resource mapping node is blocked by a potentially lower priority thread. However, at most one node per thread in the system can be blocked. To decrease the amount of memory, needed to signal the root overrun, a root-overrun token can be used instead of the mapping node. The root overrun token is linked to the mapping node and stores the reference count, root-overrun flag and a reference cache (to allow *page-reference-information retrieval*). The mapping node points to this root-overrun token. Instead of setting the flag in the mapping node, the *unmap* operation that removes the node from the mapping-database tree sets the flag in the root-overrun token. This allows the *unmap* operation to deallocate the mapping node and free up the memory space it requires.

Root-overrun tokens reduce the amount of memory that has to remain allocated by an amount that is the difference in size between the mapping node and the token.

### Consistency of the Token-Based Preempted-Thread List

Instead of per token locks in the preempted thread list, this would require to acquire the locks of at least four tokens to move the list, we propose to protect the preempted-thread list with the same lock that protects the mapping node.

If *node-locks* would have been chosen for protecting the *mapping database* nodes, each token of the *token-based preempted-thread list* would have to store the *node-lock* and moving the list would require to update the locks of the tokens in the list as well, leading to a worst case update performance of order number of threads in the system. For a single page size mapping database, the *frame-lock* does not change when moving the *token-based preempted-thread list* in the tree. See below for multiple hardware page-size support.

#### 4.3.4 Summary

The design of a preemptable unbounded-priority-inversion free mapping-database for a single hardware page-size was presented in the last section.

We introduced three operation-oriented representations for the mapping-database tree that allow post-order traversal of subtrees to unmap: *LL*, *LL-Tree* and *LL-O1*. *LL-Tree* and *LL-O1* both trade space and time to update additional pointers for a potential gain in finding the first leaf node faster. Experimental evaluation has to prove whether this tradeoff pays.

Roll-forward techniques in combination with scheduler-conscious locking were proposed to ensure data-structure consistency. Locks are proposed to be taken at the granularity of the frame, assuming low contention on the frame. A roadmap to tackle higher contention cases is included as well.

Forward progress is guaranteed by roll forward and by tracking the restart points of preempted unmap operations. To efficiently update the restart points, the *token-based preempted-thread list* was proposed, that in combination with *root-overflow detection* allows to update the restart points without having to update information in each preempted thread's TCB.

Open issues that remain are:

- Cost of *address space modifiers*
- Choice of tree representation
- Worst case overhead added to interrupt handling when rolling forward

- Multiple hardware page-size support

The last issue is discussed in the next section.

## 4.4 Multiple Page-Size Support

This sections extends the design of the single hardware-page-size mapping-database to support multiple hardware page-sizes. In the *Recursive Virtual Address Space Model*, this may lead to the following three special cases of the *address space modifiers*:

- *Split mapping*
- *Partial unmap*, and
- *United mapping*

As mentioned before, *United mappings* are not tackled in this thesis.

To minimize the space requirements for the initial address space,  $\sigma_0$  mapping nodes are mapped with the largest possible frame size. Smaller derived mappings can be requested and lead to split mappings.

### 4.4.1 Split Mapping

*Split mappings* occur if the fpage to *map* selects only a part of a large page. In this case, the selected part of virtual memory is mapped as a set of smaller pages. The smaller pages have to fit in size into the fpage.

The *map* operation maps from a larger page to a smaller page. Therefore, it selects the corresponding part of the page-frame the large page translates to and inserts a translation to this smaller part into the page tables of the target address-space. The modifications on the *mapping database* are the same as for a single hardware page-size. A child mapping node representing the smaller page is added to the mapping-database tree and linked as a child to the node of the larger page.

### 4.4.2 Partial Unmap

*Unmap* revokes access from all directly or indirectly derived mappings, including all derived *split mappings*. *Partial unmap*, however, revokes access only from those part of the larger page, that was specified by the fpage to *unmap*.

Assume a *partial unmap* of the region of virtual addresses in the interval  $[v_b, v_c]$ . The region is within a large page in the source address-space that maps the addresses in the interval  $[v_a, v_d]$ . *Partial unmap*, therefore has to revoke access from all pages that directly or indirectly received the interval  $[v_b, v_c]$  from a thread in the source address-space.

Access from pages that in addition to receiving  $[v_b, v_c]$  received also parts of the intervals  $[v_a, v_d] \setminus [v_b, v_c] = \{[v_a, v_b), (v_c, v_d]\}$  can be revoked from the entire page. Alternatively

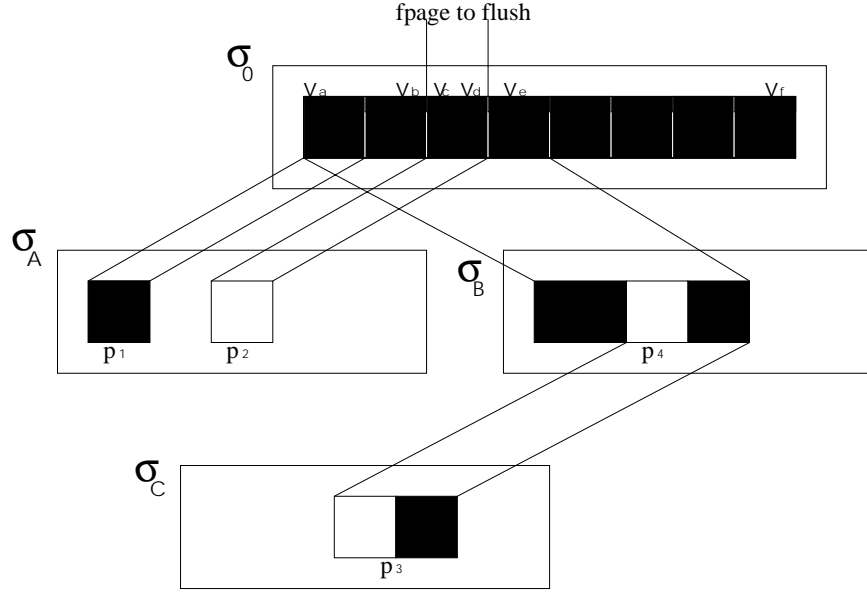


Figure 4.25: *Split-up of a large page on partial unmap.*

we can split up those pages into pages that relate to the first interval and pages that relate to the second interval. In this case, access is revoked only from the first set of pages (see Figure 4.25).

### Partial Unmap with Large Page Split Up

*Partial unmap* requires the following preprocessing before revoking access from a derived page  $p$  (see Figure 4.25):

1.  $p$  is completely derived from the addresses in  $[v_a, v_b)$  or  $(v_c, v_d]$ . In this case,  $p$  is not derived from the fpage to revoke, i.e.  $p$  needs not to be unmapped. No preprocessing is required ( $p_1$ ).
2.  $p$  is completely derived from the addresses in  $[v_b, v_c]$ . This case requires no preprocessing either. Access has to be revoked from  $p$  ( $p_2$ ).
3.  $p$  is derived from addresses in  $[v_b, v_c]$ , but also from some of the addresses in  $[v_a, v_b)$  or  $(v_c, v_d]$ , but not from both. In this case,  $p$  needs to be split up in two parts. The first one, being the set of pages that are entirely derived from  $[v_a, v_b)$  (or  $(v_c, v_d]$ ). Access from those pages need not to be revoked. The second one, being the set of pages that are entirely derived from  $[v_b, v_c]$ . Access is revoked from the second part ( $p_3$ ).
4.  $p$  is derived from addresses in  $[v_b, v_c]$ , but as well from parts of the addresses in

$[v_a, v_b)$  and  $(v_c, v_d]$ . This case is handled similar to case 3 except, that  $p$  has to be split in three parts and the two outer parts can be skipped ( $p_4$ ).

Alternatively to splitting up the page  $p$  in case 3 and 4, the entire large page can be revoked. If however, the entire page is revoked, all mappings derived from that page have to be revoked. This includes also split mappings that would fall into the cases 1 and 2.

#### 4.4.3 Root Array Structure

Skoglund et. al [DSU] proposed the *root array structure* to separate the mapping-database trees of mapping nodes corresponding to large pages from those subtrees with nodes corresponding to small pages.

The *root array structure* connects a mapping node to all subtrees of split mappings derived from the node. The root array nodes thereby contain the required pointers to step down into the subtree. Those pointers are dependent on the representation chosen:

- *LL* stores only the *map-link* in the root array.
- *LL-Tree* stores the *map-* and *child-link*.
- *LL-O1* stores the *map-* and *down-link*.

We expect, that most nodes in the tree will not have split mappings. This is trivially the case for those nodes of pages of the smallest hardware page-size. Those pages cannot be further split up. But we expect it also for larger hardware page-sizes. Motivated by that, we propose, an additional node: the *dual node*. The *dual node* is linked in between the mapping node having the split mappings and the next mapping node in *unmap* traversal direction. The *dual node* connects the mapping tree of the large page size with the *root array structure*. This saves the space required to store the link to the *root array structure* in mapping nodes without split mappings.

The *root array structures* support the search for the root-array node, given the address the split up part of the frame starts with and its size. This root-array node contains the information to step down into the subtree of split mappings.

Skoglund et. al [DSU] proposes an array based structure similar to multi-level page-tables, Szmajda [Szm99] instead, applies an LPC tree to store the root array structure in. The choice for an appropriate structure to representing the root array structure has to incorporate both the likelihood of split mappings and their distribution, i.e. whether only a few parts of the large page have been mapped or whether almost all possible parts have been mapped as split mappings.

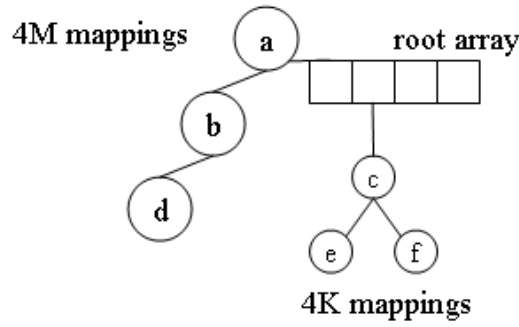


Figure 4.26: *Root array structures separate the subtrees of nodes corresponding to different page sizes.*

We expect few split mappings in the system at all. Pages are usually split once, and then the smaller page size is used. The question on the distribution cannot be answered such easily and requires further examinations. An observation of only a few parts of the large page split up would preference the LPC approach, if instead most or all parts are split up, the array based approach is preferable.

We implemented the array based approach.

The root array structure has the property of separating subtrees of nodes corresponding to pages of different sizes.

#### 4.4.4 Frame Locks Revisited

*Frame-locks* protect the mapping database tree structures to ensure data-structure consistency. While the single page-size mapping-database requires only to acquire a single *frame-lock*, multiple differently sized frames can be present in the mapping-database tree of the multi page-size database.

There are two alternative solutions:

**Frame lock for largest frame size:** A single *frame-lock* is assigned only to the largest frame (corresponding to the  $\sigma_0$  mapnode). This *frame-lock* protects the entire subtree, including split mappings with a smaller page-size.

**Frame lock per frame size:** Alternatively a lock per frame size per frame can be used. The lock protects only the frame of the corresponding size.

A single *frame-lock* for the largest frame allows to apply the same locking algorithm as presented for the single page-size mapping-database. However, concurrent operations that operate on different smaller frames are mutually excluded, though they can be

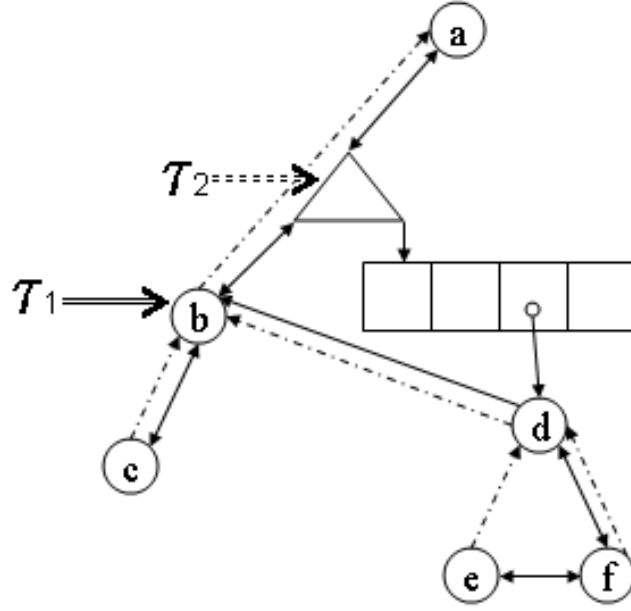


Figure 4.27: *Tree representation LL for multiple hardware page sizes. dual nodes (triangles) hook the root array structures into the mapping tree.*

processed in parallel. Second, the largest page size of some architectures can be huge (the largest page size of the Intel Itanium is for example 256 MB). The assumption of having low contention is unlikely to hold for such large frame sizes.

The second solution allows for acquiring smaller locks. Operations on large page sizes, however, have to mutually exclude all operations on the corresponding smaller frames. And therefore need to acquire multiple locks. This is both slow and difficult to avoid deadlocks in.

Instead, we introduce *root array structures* (see above) to separate the subtrees related to different frame-sizes. This allows to acquire a single *frame-lock*, corresponding to the frame and frame-size of the node that is processed. The *frame-lock* of the node of the larger page protects the *root array structure* as well. When modifying the *root array structure* while processing the node, we have to have to acquire the larger frame-lock as well.

Figures 4.27, 4.28 and 4.29 show the three tree representations for the multi page size mapping database. For the reason of simplicity, we omit the page-table links and the links to the *token-based-preempted-thread-lists* from *LL-Tree* and *LL-O1*. Section 4.4.5 reasons about the additional *token-based-preempted-thread-list* at the dual node (see below).

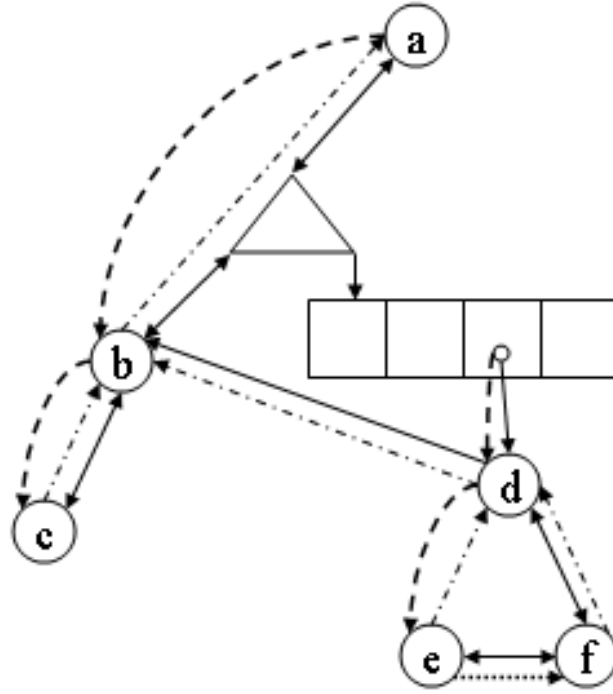


Figure 4.28: *Tree representation LL-Tree for multiple hardware page sizes.*

### Traversal on Unmap

The traversal algorithm for the *unmap* operation has to be modified for a multi page-size mapping-database. In particular, the *root array structures* need to be incorporated to traverse through the subtrees containing split mappings.

We subdivide the traversal path of *unmap* into two sweeps:

- *traverse down* searches for the first leaf node of a subtree and traverses into the *root array structure*,
- *traverse up* sweeps through the nodes in post order direction and revokes address from the corresponding pages.

The same constraints apply for the multi-page-size traversal-path as for the single-page-size path. In particular we have to require that all nodes in the subtree are processed, and that the traversal path can be traversed iteratively. Figure 4.30 shows the modified traversal path of the *unmap* operation.

*Traverse down* starts from the root node of the subtree of nodes to process. It searches the first leaf node of this subtree. Thereby, it does not traverse down into the *root ar-*

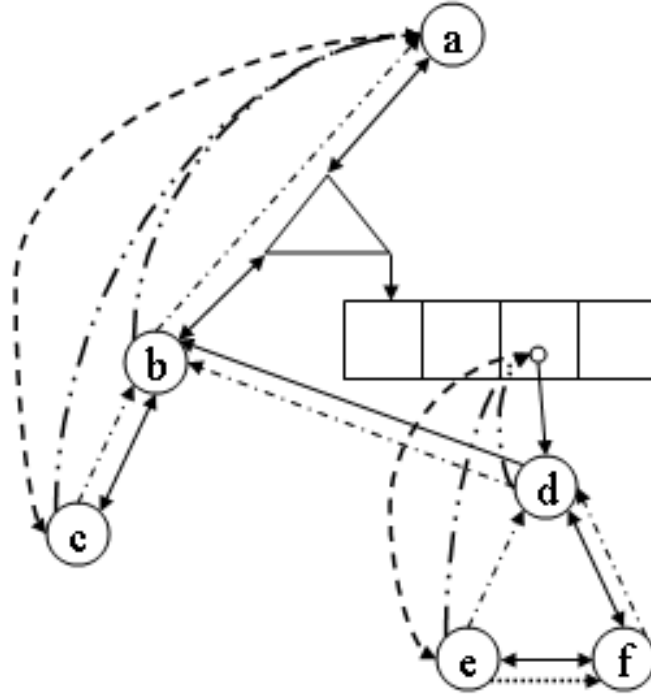


Figure 4.29: *Tree representation LL-O1 for multiple hardware page sizes.*

*ray structures* of the nodes passed. If the first leaf node has split mappings, it revokes access from the page of the node. However, the node itself remains linked into the tree. It then traverses down into the *root array structure* for this leaf node, searching for the root array node that links the subtree of the smallest size with and that corresponds to the lowest frame address. After that, *traverse down* continues to search for the first leaf of this subtree.

If a first leaf is found that has no split mappings, the traversal algorithm switches to the *traverse up* sweep.

*Traverse up* removes access from the page the node corresponds to and continues with the next node in post order direction. If this node contains split mappings, the traversal algorithm switches back to *traverse down* and searches for the root array node and the first leaf node of the corresponding subtree. *Traverse up* also returns to *traverse down* after it has processed the last node of a subtree of split mappings. In this case, it searches the *root array structure* for the root array node with the next larger frame address and *traverses down* into the subtree linked to that root array node. If no such root array node is found, it increases the frame-size to look for and searches for the root array node with the smallest frame address of this size. Once, all split mappings have been processed, *traverse up* finally removes the root array structure and the mapping

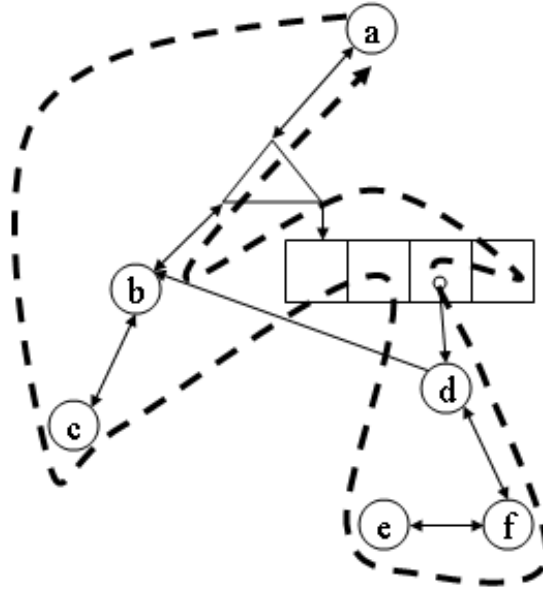


Figure 4.30: The traversal path taken by *unmap* in a multi page tree representation (here *LL*). The red dotted line marks the path taken.

node containing the split mappings if this is necessary. After that, *traverse up* continues with the next node in post order direction until reaching the root node.

The revocation of access rights before *traversing down* into the *root array structure* is required, because otherwise, split mappings could be generated in the already traversed subtrees that relate to pages that still have the access right that the *unmap* operation revokes. If this is not done, those pages would not be traversed by *unmap*.

Note, special activity is required for *restart-point tracking* when *traversing up* from the split mapping node to the node of the large page (see below).

#### 4.4.5 Restart-Point Tracking

Multiple hardware page-sizes complicates *restart-point tracking*. The *restart-point tracking* when removing the last node of a subtree of split mappings requires special case handling. Furthermore, the *frame-lock* protecting the *token-based preempted-thread list* changes in this case.

### Restart-Point Tracking for Split Mappings

The threads that have been preempted in a subtree  $S$  of split mappings have to restart with searching the next root array node to *traverse-down* when the subtree  $S$  has been completely removed. However, if we would merge the *preempted-thread-list-tokens* of those nodes into the *token-based-preempted-thread-list* of the node  $n$  of the large page-size, a *map* operation might propagate those tokens to a child node of  $n$ . In this case, a restarted *unmap* has to process the child node of  $n$  and the already completed subtrees a second time. Forward progress cannot be guaranteed.

To avoid this situation, we introduce a second *token-based-preempted-thread-list*, the *split-token-based-preempted-thread-list*. In the above case, the tokens are propagated into the *split-token-based-preempted-thread-list* instead of the *token-based-preempted-thread-list*. *Map* updates the restart points of threads in the *token-based-preempted-thread-list*, but omits the updates for the *split-list*. An *unmap* operation removing the node, merges the two lists and updates the *restart-points* to point to the next node to process.

Note, tokens are never propagated down into a subtree of split-mappings.

### Frame-Locks and Restart-Point Tracking

Frame locks change when propagating to a node with a larger page size. To ensure the consistency of the *token-based-preempted-thread-lists* we store a link to the *frame-lock* to use with the nodes. However, this pointer has to be updated in the situation described above. Therefore, the tokens in the list have to be traversed and the locks in them have to be updated. To avoid having to atomically update the entire list, we propose the following algorithm:

Before removing the last node of a subtree of split mappings, the *token-based-preempted-thread-list* of this node is traversed and each token is moved into the *split-token-based-preempted-thread-list* after the link to the lock was updated. This operation is preemptable after having processed a single token. If the updating thread gets preempted, it links its TCB to a token in the *split-token-based-preempted-thread-list*. An *unmap operation* of a thread that is restarted and finds its TCB linked to the *split-token-based-preempted-thread-list* first checks for a pending lock update before resuming its operation. If this is the case, it continues to update the tokens.

#### 4.4.6 Space Requirements

The three tree representations *LL*, *LL-Tree* and *LL-O1* differ in the number of pointers they have to store. In addition to that, the mapping node has to store the *payload* information and the two links to the *root-overflow token* and the *token-based-preempted-*

Representation	Size of the Mapping node
<i>LL</i>	8 x 32 bit words
<i>LL-Tree</i>	8 x 32 bit words
<i>LL-O1</i>	10 x 32 bit words
<i>LL-opt</i>	6 x 32 bit words
<i>LL-O1-opt</i>	8 x 32 bit words

Table 4.4: *The space requirements of a mapping node in the different tree representations.*

*thread-list*. We allocate 27 bits for the pointers, i.e. more than 130 million mapping nodes can be dereferenced. The actual layout of the data structures is shown in Appendix A.

Table 4.4 summarizes the size of the mapping node structures. The two representations *LL-opt* and *LL-O1-opt* are special optimizations for the Intel IA-32 architecture. Certain assumptions on the layout of virtual address space of the L4  $\mu$ -kernel<sup>2</sup> allowed for the reduction of the space required. The size of a cacheline of the Intel IA-32 architecture is 32 bytes.

## 4.5 Summary of the Design

The last section introduced the proposed design of a preemptable unbounded-priority-inversion-free mapping-database. The *mapping-database* is designed to support multiple hardware-page-sizes.

We introduced three operation oriented representations for the mapping database tree that allow post order traversal of subtrees to unmap: *LL*, *LL-Tree* and *LL-O1*. In *LL-Tree* and *LL-O1* space and update time of additional pointers is traded against a potential gain in searching for the first leaf node to start an *unmap* operation from. Subtrees of nodes corresponding to different hardware page sizes are separated with root arrays that are linked to the tree with *dual nodes*.

Roll-forward techniques in combination with scheduler-conscious-locking were proposed to ensure data structure consistency. The *mapping-database* trees are protected by *frame-granular* locks. Such a lock exists per frame and per possible frame size.

*Root-arrays structures* separate subtrees with mapping nodes that correspond to pages of a different size. This allows to acquire only those *frame-locks* of the frame-sizes operated on. The *root-array structures* are protected by the larger *frame-lock*.

Forward progress is guaranteed for all operations. We proposed roll forward and *restart-point tracking* to guarantee progress. We proposed a new method to efficiently update

<sup>2</sup>On Intel IA32 architectures, the virtual address space of the L4  $\mu$ -kernel is mapped into the upper most gigabyte of all address spaces.

the *restart-points*: the *token-based-preempted-thread-list*. In combination with *root over-run detection* this list allows to update the preemption points without having to traverse the lists.

The following issues remain open from the design:

- Cost of *address space modifiers*
- Choice of tree representation
- Worst case overhead added to interrupt handling when rolling forward



## Chapter 5

# Experimental Results and Analysis

This chapter analyses the performance of the *address space modifiers* implemented with the three proposed tree representations: *LL*, *LL-Tree* and *LL-O1*. It compares those representations with the non-preemptable implementation of Pistachio and to the preemptable implementation in the Fiasco  $\mu$ -kernel. Furthermore, it estimates the time, interrupt handling is delayed when rolling forward the *address space modifiers*.

### 5.1 Experimental Setup

The *mapping database* was implemented in the L4Ka Pistachio  $\mu$ -kernel<sup>1</sup>. The Pistachio  $\mu$ -kernel is a portable implementation of the experimental L4-Version X.2 API [Tea].

The experiments were implemented in an adapted version of the Nutcracker library [Elb]. Originally suited for black box testing of the L4  $\mu$ -kernels, the Nutcracker provides library functions that allow for an easy setup of test and experiment code.

The measurements are performed on a 500 MHz Intel Pentium III processor with a 512 KB shared level 2 cache, and separate 16 KB L1 caches for code and data. All caches are 4 way set associative. The size of a cacheline is 32 bytes.

The Intel Pentium III supports 4KB pages as well as 4MB pages. It has separate TLBs for data and code pages. The data TLBs have 64 entries for 4KB pages and 8 entries for 4MB pages. The instruction-TLBs have 32 entries for 4KB and 2 entries for 4MB pages.

L1 hits on this processor take 2 cycles (the L1 cache can handle two requests in parallel), L1 misses hitting in the L2 cache take approximately 10 cycles, missing in the L2 takes about 50 cycles – those numbers assume a valid translation for the memory

---

<sup>1</sup>Available at <http://l4ka.org>

access to be cached in the TLB.

For the experiments we start two threads in two different address spaces:  $\tau_1$  and  $\tau_2$ . During the experiments only those threads run and only one at a time. When acting as a receiver, one thread accepts mappings. The corresponding other thread acts as the *mapper*, sending the pages. Time is measured with the Intel Pentium cycle counter (`rdtsc`).

The Pistachio kernel is by the time this thesis was written about to be ported to multiprocessor systems. Therefore, all measurements were performed on a uniprocessor system. Because we applied well known multiprocessor synchronization primitives, we do not expect unforeseen behavior when conducting the multiprocessor measurements.

## 5.2 Performance of the Address Space Modifiers

This section evaluates the performance of the *address space modifiers*: *map* and *unmap* for the three proposed database representations. Further it compares them with the mapping database in Pistachio and in Fiasco.

*Grant* is omitted because it does not modify the mapping database structure. The measurements are performed with warm caches to retrieve best case performance numbers.

### 5.2.1 Map Performance Comparisson

The first experiment measures the performance of resolving a page fault by mapping a 4KB page. The thread  $\tau_2$  has been configured to generate pagefaults by writing to a page in its address space that is not mapped. The pagefault is caught by the  $\mu$ -kernel and translated into a pagefault IPC message to  $\tau_2$ 's pager:  $\tau_1$ .  $\tau_1$  replies with *mapping* a 4KB page that it requested from  $\sigma_0$  beforehand. Always, the same page is mapped.

We measure the execution time of the replying map operation. The measurement is started before  $\tau_1$  sends the *map*-IPC and it is stopped after  $\tau_2$  resumed the operation that caused the page fault.

In addition to that, we measured the pure mapping-database performance for our proposed operations. This is the time required to insert the new mapping node into the *mapping-database* tree.

Figure 5.1 shows the execution times (in cycles) measured. It shows those values in a comparison of the three tree representations: *LL*, *LL-Tree* and *LL-O1* with the non-preemptible mapping database of Pistachio and with the mapping database of Fiasco. Furthermore, it shows the results of the optimized versions of *LL* and *LL-O1* for the Intel Pentium Processor: *LL-opt* and *LL-O1-opt*.

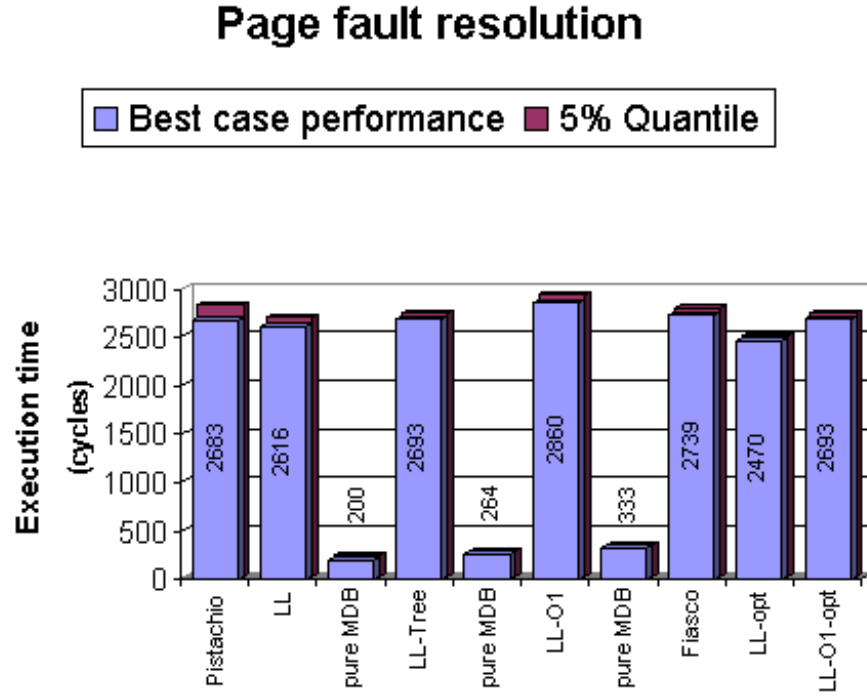


Figure 5.1: Performance comparison of mapping a 4KB page to resume a page fault.

As expected *LL* outperforms *LL-Tree* by 64 cycles and *LL-O1* by 133 cycles (pure mapping database performance). This results in *LL-Tree* being about 32% slower, *LL-O1* about 67% slower than *LL*. Compared to the costs of the entire operation, this difference is less than 5%.

The performance differences can be explained with the additional pointers that have to be updated in the more complex representations.

Fiasco is faster than *LL-O1* but slower compared to *LL* and *LL-Tree*. The optimized version *LL-opt* clearly outperforms Fiasco's map operation by 264 cycles (9.8% of the entire operation costs).

Unexpectedly, Pistachio performs a little worse than *LL*. It is outperformed by *LL* by an amount of 67 cycles. *LL-opt* outperforms Pistachio by 213 cycles (7.9%). We expect that a comparable performance to *LL-opt* can be achieved by further optimizing the implementation in Pistachio.

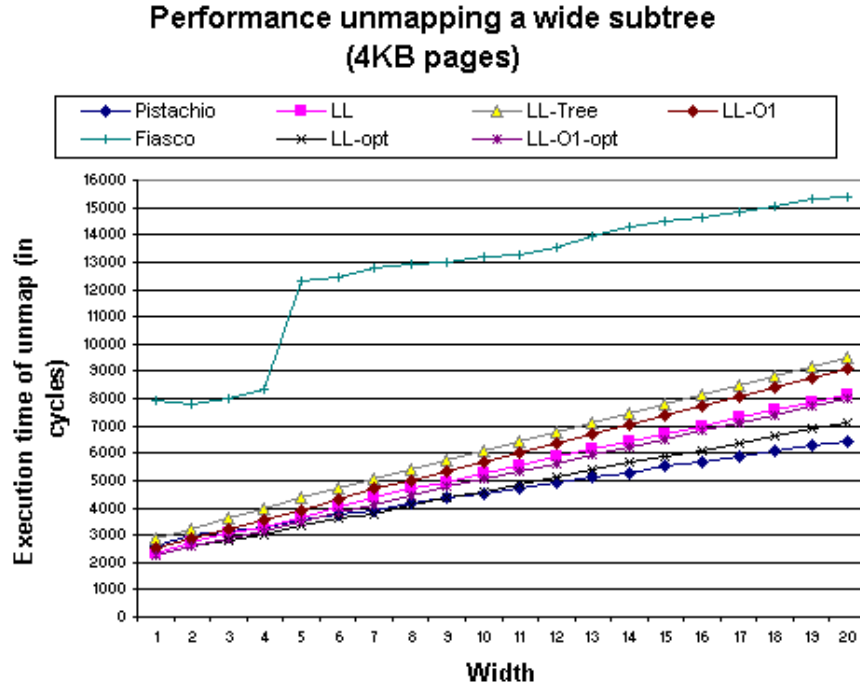


Figure 5.2: *Performance of unmapping a wide subtree of 4K pages.*

### 5.2.2 Unmap Performance Comparisson

The second experiment measures the performance of the *unmap* operation.

The tree structures *LL-Tree* and *LL-O1* have been proposed to speed up the performance of finding the first leaf node. *LL* finds this first leaf node in order  $n$  traversal steps, whereby  $n$  is the number of nodes in the tree. *LL-Tree* requires only order depth of the subtree traversal steps. *LL-O1* is capable to find the first leaf node in order one. To corroborate the choice for one of those structures, we measure the execution time of the *unmap* syscall while it revokes access from a wide and from a deep subtree. We vary the number of mappings  $n$  in the subtree.

In addition to the modification of the mapping database, the results include the time to enter and exit the kernel, to modify the page tables and to invalidate the TLB entries if this is required.

Figure 5.2 shows the execution time of the *unmap* syscall as a function of the depth of the subtree, Figure 5.3 shows the execution time as a function of the width of the subtree. As can be seen, the performance increases linearly with the number of nodes in the subtree. The performance decrease from unmapping 4 mappings to unmapping 5 mappings is because Fiasco starts to reallocate the mappings nodes into a smaller array. To accomplish that, it needs to copy the nodes.

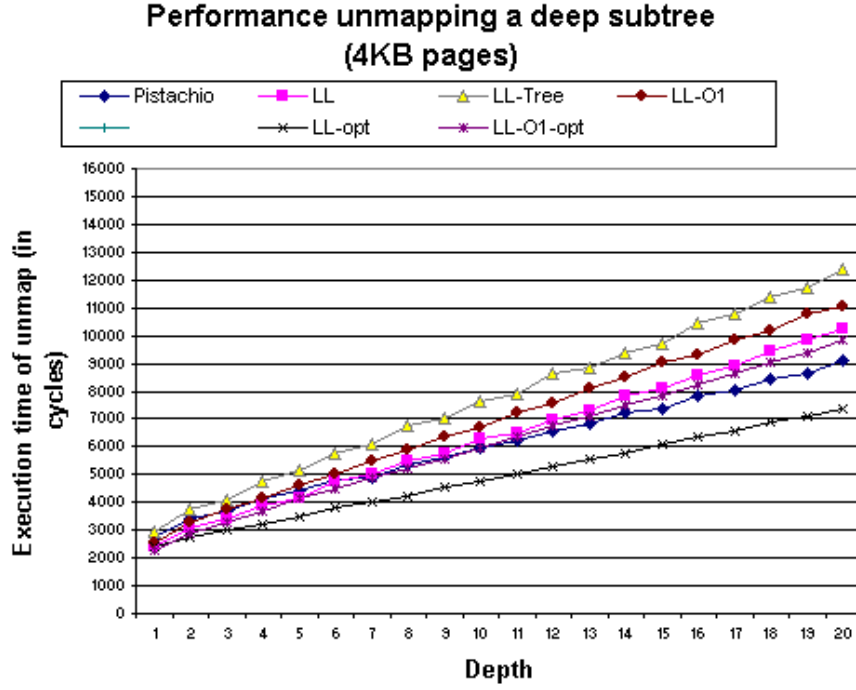


Figure 5.3: *Performance of unmapping a deep subtree of 4K pages.*

With the depth increasing, *LL-O1* performs better than *LL-Tree*. The fewer pointers that have to be updated for *LL* makes it outperform both, *LL-Tree* and *LL-O1* by 2149 cycles = 19.5% (824 cycles for *LL-O1* = 8%) at a depth of 20 and by 1356 cycles = 16.6% (967 = 11.9% for *LL-O1*) at a width of 20 .

The expected performance gain of *LL-O1*, being able to find the first leaf in order 1, is eliminated by the time to update the additional pointers. The optimized version *LL-O1-opt* shows the desired effect, but is still outperformed by *LL-opt* by an amount of about 12.0% (at width 20).

A comparisson of the non-preemptable mapping database implementation in Pistachio with *LL* shows that the Pistachio performs better. The difference is 12.6% for the deep subtree and 26% for the wide subtree. The optimized version *LL-opt* significantly outperforms Pistachio for deep mappings and achieves a performance that is only 10.1% slower than Pistachio for wide mappings.

The *unmap* operation of Fiasco is significantly outperformed by all other representations, including Pistachio. The graph increase more slightly for up to four measure-

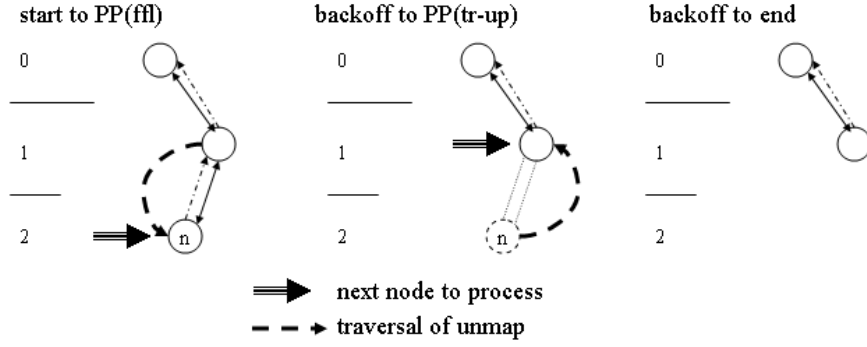


Figure 5.4: The mapping database tree of the unmap operation at the three points, unmap is preempted.

ments, however the base costs are significantly larger.

We performed the same two experiments for 4MB pages. The results were similar as for 4KB mappings. Therefore, we did not include the graphs into this thesis.

### 5.3 Interrupt Latency

Interrupt handling is delayed for the time, a mapping database operation is rolled forward. We estimate this delay by measuring the time, mapping database operations are rolled forward. Prior to starting the measurement we invalidate all caches and translation lookaside buffers. This prevents the mapping database operations from improving their performance with cached code and data. The performance results of those operations estimate worst case performance.

In a *map* operation, the insertion of the new mapping node and the page table update is rolled forward. *Unmap* contains more preemption points. For this measurement, *unmap* removes a single mapping node. It is possible to preempt this operation at two points (see Figure 5.4). The first preemption point  $PP(ffl)$  allows for a preemption after finding the first leaf, but before processing it<sup>2</sup>. The second preemption point  $PP(tr - up)$  allows for a preemption after the derived mapping is removed, but before processing the root node of the tree<sup>3</sup>. We measure the time the unmap operation is rolled forward, i.e. from the start of the unmap to a preemption at  $PP(ffl)$ , backing off from this preemption to the next preemption at  $PP(tr - up)$ , and finally backing off from the second preemption to the end of the roll forward. This end is immediately

<sup>2</sup>In general, the find first leaf preemption point allows to preempt the unmap after having stepped one node further in the direction of the first leaf.

<sup>3</sup>The traverse up preemption point allows to preempt the unmap after a node is completely processed, but before processing the next node.

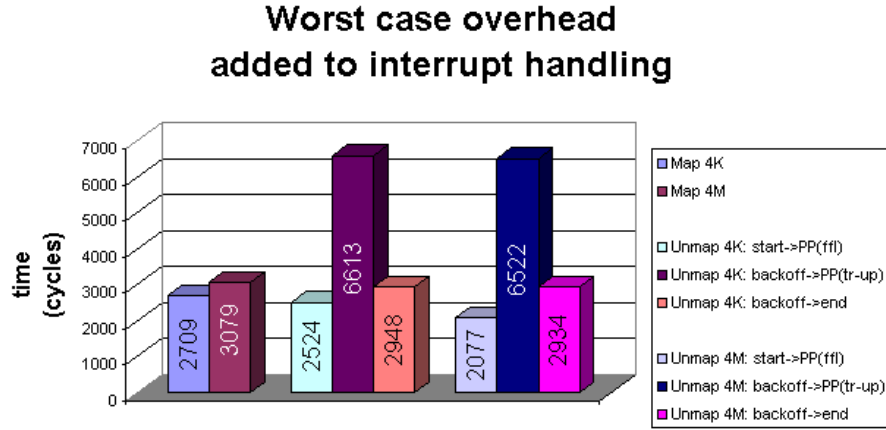


Figure 5.5: Overhead added to interrupt handling by the mapping database operations *map* and *unmap*.

before the *unmap* operations returns.

Figure 5.5 shows the results of those measurements. As can be seen, *map* adds an overhead of 2709 cycles for 4KB mappings and 3079 cycles for 4MB mappings to interrupt handling. This corresponds to  $5.4 \mu s$  and  $6.2 \mu s$ . *Unmap* adds an overhead of 6613 cycles for 4KB and an overhead of 6522 cycles to 4MB mappings. This corresponds to  $13.2 \mu s$  and  $13.0 \mu s$ .

## 5.4 Analysis

Corroborated by those results, the choice for one of the three representations clearly has to be in preference of *LL* and in particular for *LL-opt* on an Intel Pentium processors. The fewer number of pointers that have to be updated by *LL* make it outperform the other proposed tree representations. The expected speed up from finding the first leaf node faster is eliminated by the overhead to update the additional pointers of the larger structures.

In comparison to the non-preemptable *mapping-database* implementation in Pistachio, *LL* shows a comparable performance for address space construction with *map*. The cost of a preemptable *unmap* operation, however, is a slowdown in performance between 12.6% and 26%.

Compared to the solution in the Fiasco  $\mu$ -kernel, all three tree representations show a

significant better performance on *unmap*. Fiasco achieves a comparable performance for *map*, however, it is clearly outperformed by *LL-opt*.

Whether the significant loss in performance is caused by the different tree representation, or whether those costs are inherent to the “helping-scheme” applied, however, cannot be seen with those measurements. An in depth analysis of Fiasco’s mapping database remains to be done.

The mapping database delays interrupt handling by at most  $13.2\ \mu\text{s}$  before its operations are preempted.

## Chapter 6

# Conclusion and Future work

This chapter concludes. Furthermore, it outlines future work and surveys possible directions, the work presented in this thesis, can be extended to.

### 6.1 Conclusion

Virtual memory is an important feature of operating systems for interactive or partially untrusted applications. However, standard virtual memory policies are often not optimally suited for specialized applications such as database-management and multimedia.

The *externalization* of virtual memory management and protection policies into application specific managers at user level is one approach to fulfill the requirements of those applications. The *Recursive Virtual Address Space Model* is such an approach to *externalization*.

The key idea of this model is to construct and modify address spaces recursively. Page-frames that the constructing address space itself has access to can be *mapped* or *granted* into an address space. The target address space thereby gets access to those page-frames. Access can be revoked from directly or indirectly mapped pages with the *unmap* operation.

Existing implementations of the *Recursive Virtual Address Space Model* show undesirable behavior or unnecessary slow and complex behavior such as:

- *long interrupt latencies*,
- *unbounded priority inversion*, or
- *complex helping-schemes* to avoid both.

This behavior has limited the applicability of the model in some systems. The goal of this thesis was to investigate this problematic behavior and to propose solutions that

avoids it.

In this thesis we identified that preorder traversal of the mapping database tree on *unmap* (see Section 2.6.3) may cause unbounded priority inversion. Roll forward of long operations and complex helping-schemes have been proposed to avoid this unbounded priority inversion. However both avoidance schemes cause undesirable side effects such as long interrupt latencies for the first and restrictions on the scheduling of threads for the second (see Section 2.6.5). In Section 2.6.4, we showed that similar problems arise for all other traversal methods except for post order traversal.

### 6.1.1 Approach

With the goal of avoiding long interrupt latencies, unbounded priority inversion and helping, this thesis presented a preemptable mapping-database design in Chapter 4. The key techniques to achieve this goal were:

- *preemptable post order traversal* of the mapping database tree,
- *roll forward in combination with scheduler conscious frame-granular locking* to ensure the consistent modifications, and
- a method to efficiently *track the restart points* of preempted operation. In combination with roll forward this allowed us to guarantee forward progress of all operations on the mapping database and thereby freeness of starvation.

We presented and experimentally evaluated three operation-oriented representations of the mapping database tree: *LL*, *LL-Tree* and *LL-O1*. For efficient restart-point tracking we proposed to publish and update the *restart-points* of all preempted operations. We introduced a new technique: the *token-based-preempted-thread-list* to efficiently update those *restart-points*. This technique in combination with *root-overflow detection* allowed us to update the *restart-points* of all threads without having to traverse the TCBs of the threads.

The mapping database supports multiple different hardware page sizes and is suited for uniprocessor- as well as small scale multiprocessor systems.

### 6.1.2 Summary of Results and Conclusions

Experimental validation has shown a clear preference for the data structure *LL*. Having less pointers to update and a smaller mapping node structure benefits even compared to requiring order ( $N$ ) traversal steps to find the first leaf. Thereby,  $N$  is the number of nodes in the subtree to process on *unmap*.

*LL-Tree* and *LL-O1* have been proposed to optimize the search for the first leaf node. *LL-Tree* requires to traverse only order ( $D$ ) nodes, whereby  $D$  is the depth of the first leaf relative to the root of the subtree to process. *LL-O1* achieves this in the best case in order (1). The expected increase of *unmap* performance, however, could not be seen. Instead, the larger data-structures and the additional pointers that have to be updated, eliminated the performance gain from finding the first leaf node faster.

The ability to roll forward to a point where it is safe to be preempted, combined with the benefits of a multiprocessor synchronization scheme that does not preempt lock-holders, was crucial in the design of the mapping database and made it possible to omit helping-schemes entirely.

We were able to place the preemption points such, that the mapping database structure is in a consistent state with all locks released when a preemption occurs. The operations are rolled forward between the preemption points. During this time, interrupt handling is delayed. *Map* delays interrupts handling by a maximum of  $6.2 \mu s$ , *unmap* delays it by a maximum of  $13.2 \mu s$ .

Systems that have to respond in a shorter time, have to apply a different solution. As long as the activity responding to the interrupt is independent of the virtual memory subsystem, the  $13 \mu s$  could be broken down by allowing this independent interrupt handler to preempt the mapping database operation more frequently. Afterwards however, the preempted mapping database operation has to be resumed prior to executing VM dependent code.

We have to admit, that the worst case execution time of the *unmap* operation can still not be determined by the unmapping thread. *Unmap* is not capable to be used by realtime threads. This is because the size of the subtree to process on *unmap* cannot be limited.

The performance measurements show that a comparable performance to a non-preemptable implementation can be achieved for *map*. The cost of preemptability comes into account for revoking access from pages with *unmap*. *Unmap* is between 12.2% and 26% slower than the non-preemptable implementation in Pistachio (The first value is for the architecture specific optimization *LL-opt*).

To sum up the conclusions:

- *Simple data structures are preferable for the mapping database.*
- *Good preemptability can be achieved without having to rely on complex helping-scheme.*
- *This can be done while preserving a performance comparable to a non-preemptable solution.*

As a general advice we have to conclude:

*Causes of unbounded priority inversion should be eliminated or at least minimized by design, instead of relying on avoidance schemes such as helping.*

## 6.2 Future Work

This section surveys future work that is required to resolve the issues left open in this thesis. Furthermore, we present possible directions, the work presented in this thesis can be extended to.

The following issues remain open and have to be resolved in future work:

- *Costs and benefits of united mappings*
- *Costs and benefits of mapping node reallocation*
- *Appropriate choice for the root array structure*

The comparison to Fiasco showed significantly worse performance compared to our proposed solution. However, in order to bring more evidence to those results, an in depth analysis has to explore whether this is caused by the different tree representation, or whether this is crucial to the helping-scheme applied.

A validation of the experimental results on a small scale multiprocessor system remains to be done. We do not expect unforeseen behavior or results.

This experimental validation has to explore the costs of the TLB shutdown algorithms. In order to make changes to the page table effective, the corresponding TLB entries have to be invalidated. In symmetric multiprocessor system, this requires to “shoot-down” the TLB entries on the other CPUs. The revocation of all directly and indirectly derived mappings allows to collect the entries that have to be invalidated before triggering the invalidation process with a cross processor interruptions.

### United Mappings

*United mappings* allow to unite multiple pages that have been mapped both physically and virtually adjacent and with the same rights into a single large page. Potentially, this increases TLB coverage and thereby leads to an increase of overall system performance. The difficulties in supporting *united mappings* in the mapping database, is that pages no longer origin from only one source page. Instead, they may origin from several source pages. The mapping database structure required to represent this relation is a graph.

## Root Array Structures

To be able select an appropriate representation of the *root array structure* application scenarios that mixes multiple page sizes have to be explored. In this thesis we assumed that split mappings are rare. Furthermore, we assumed that if a page has derived split mappings, that almost all possible parts of the page are in use and have been mapped to other address spaces as split mappings. Therefore, we selected an array based *root array structure*.

## Mapping Node Reallocation

The possibility to reallocate mapping nodes to a different memory location might be beneficial for some scenarios (see Section 4.1.2). In this thesis we discussed the required pointer updates that have to be performed when reallocating mapping nodes in *LL*, *LL-Tree* and *LL-O1*. We estimated the costs of those updates, however a cost benefit analysis of reallocation is still outstanding.

## Extensions

We see the possibilities to extend the work presented in this thesis in the following three directions: large scale multiprocessors, external kernel resource management and realtime capable unmap.

### Large Scale Multiprocessors

The solutions in this thesis were targeted for small scale multiprocessor systems. In order for the *Recursive Virtual Address Space Model* to be applied on larger scale multiprocessors, further work is required. We expect, that the assumptions concerning the expected level of contention on the locks will differ. Furthermore, the access times to memory are not necessarily uniform in large scale multiprocessor systems. Therefore, the locality of memory has to be considered for the design.

### External Kernel Resource Management

Currently, the L4  $\mu$ -kernel manages the kernel resources it allocates itself. This is of course not desirable because it puts policy in the  $\mu$ -kernel. This policy might not be optimally suited for all subsystems envisaged to run on top of the  $\mu$ -kernel.

Haeberlen [Hae01] proposed an extension to the Recursive Virtual Address Space Model to allow for kernel resource management at user level. The key idea of this approach is that whenever the kernel requires a resource that is not available, a resource fault is generated. Similar to page faults, the resource fault is translated into a message to a pager at user level. To resolve this resource fault, the pager *maps* a page to the  $\mu$ -kernel. Prior to using this page, the kernel upon receiving this page prevents all

threads in the system from accessing it. The important restriction is, that the pager cannot map arbitrary pages to the kernel. Instead, only pages that it received with a special access right: the k-right, can be mapped. The threads that mapped the page directly or indirectly to the pager implicitly agreed that this k-page might be used for kernel resource management and if so, that it is no longer accessible by them.

### Realtime Capable Unmap

The Recursive Virtual Address Space Model does not threads mapping a page to limit the number of mappings that can be derived from this page. Because of this, the execution time of *unmap* cannot be determined a priori. *Unmap* is not capable to be used by realtime threads.

An extension of the model, for example *map-quotas* would allow to limit the number of derived mappings. However, further work is required to explore appropriate extensions.

# Appendix A

## Data Structures

This chapter presents the layout of the data structures used for the three tree representations: *LL*, *LL-Tree* and *LL-O1* in the mapping database.

### A.1 Mapnode

The **mapnode** data structure stores the mapping nodes of the mapping database.

The **mapnodes** of *LL*, *LL-Tree* and *LL-O1-opt* are aligned in memory to 32 bytes, the **mapnodes** of *LL-O1* and *LL-opt* to 8 bytes.

*LL*:

map link <sub>(28)</sub>		ac <sub>(3)</sub>	m	$w_0$
unmap link <sub>(28)</sub>		ai <sub>(3)</sub>	u	$w_1$
parent link <sub>(27)</sub>		(5)		$w_2$
space <sub>(19)</sub>	depth <sub>(13)</sub>			$w_3$
pg table entry <sub>(30)</sub>			(2)	$w_4$
tb preemption lst <sub>(28)</sub>		P <sub>(4)</sub>		$w_5$
root overrun <sub>(30)</sub>			g l	$w_6$
dummy <sub>(32)</sub>				$w_7$

**map link:** Pointer to the next node. New mappings are inserted in this direction.

**unmap link:** Pointer to the previous node. The unmap link of split mappings point to the node of the large page.

**parent link:** Pointer to the parent node. Again, the parent link of split mappings points to the node of the large page.

**m:** Set if the map link points to a dual node, i.e. the next node contains split mappings.

**u:** Set if the unmap link points to a dual node, i.e. this node contains split mappings.

**depth:** The depth of the node in the tree, i.e. the distance to the  $\sigma_0$ -mapnode.

**space:** Address space the mapping is established in.

**pg table entry:** Pointer to the page-table entry. The virtual address can be derived by xor-ing the value of the shadow page-table with the address of the mapnode.

**p:** Index into an array of supported hardware page-size. The array contains the page-sizes sorted by size.

**ac:** A cache for the access rights to be mapped. This field is required to simulate page reference information on architectures without hardware set reference bits.

**ai:** The reference cache.

**tb preemption lst(ptl):** Pointer to the head of the *token-based-preempted-thread-list*.

**root overrun:** Pointer to the root-overrun token.

**l:** Lock update pending.

**g:** Grant reallocation pending.

**dummy:** The dummy is used to align the mapping node of *LL* to 32 bytes.

*LL-Tree:*

map link (28)	ac (3)	m	$w_0$
unmap link (28)	p (4)		$w_1$
space (19)	u	ptl <sub>1</sub> (12)	$w_2$
pg table entry (30)		ptl <sub>2</sub>	$w_3$
parent link (27)		ptl <sub>3</sub> (5)	$w_4$
sibling link (27)	ai (3)	l g	$w_5$
root overrun (30)		ptl <sub>4</sub>	$w_6$
child link (27)		ptl <sub>5</sub> (5)	$w_7$

**child link:** Pointer to the left most child node.

**sibling link:** Pointer to the right sibling of the mapnode.

**ptl:** The link to the *token-based-preempted-thread-list* is split into 5 parts  $ptl_1$  -  $ptl_5$ .

*LL-O1:*

map link <sub>(29)</sub>	(2)	m	$w_0$
unmap link <sub>(29)</sub>	(2)	u	$w_1$
parent link <sub>(29)</sub>	ac	(3)	$w_2$
sibling link <sub>(29)</sub>	ai	(3)	$w_3$
space <sub>(19)</sub>	(13)		$w_4$
pg table entry <sub>(30)</sub>	g	l	$w_5$
preemption lst <sub>(28)</sub>	p	(4)	$w_6$
up link <sub>(29)</sub>	(2)	r	$w_8$
down link <sub>(29)</sub>	(3)		$w_7$
root overrun <sub>(30)</sub>	(2)		$w_9$

**down link:** Pointer to the first leaf node, null if a common root mapnode exists that shares the same first leaf.

**up link:** Points to the common root mapnode that shares the first leaf.

**r:** Up points to a root array node if this flag is set.

*LL-opt:*

map link (29)					ac (3)	$w_0$
p (4)	0xF + unmap link (25)				ai (3)	$w_1$
ptl1 (8)		0xC + space (17)			depth (7)	$w_2$
ptl2 (4)	0xC + pg table entry (28)					$w_3$
ptl3 (4)	u	m	0xF + parent link (26)			$w_4$
ptl4 (8)	g	l	0xF0 + root overrun (22)			$w_5$

0xF, 0xC and 0xF0 denote the most significant bits that are assumed, 0xF0 means 8 bits are fixed: 11110000, 0xC fixes only the upper two bits to 11, 0xF the upper four: 1111.

*LL-O1-opt*:

map link <sub>(28)</sub>			ai <sub>(3)</sub>	m	$w_0$
p <sub>(4)</sub>	0xF + unmap link <sub>(24)</sub>		ai <sub>(3)</sub>	u	$w_1$
ro1 <sub>(15)</sub>		0xC + space <sub>(17)</sub>			$w_2$
ptl1 <sub>(4)</sub>	0xC + pg table entry <sub>(28)</sub>				$w_3$
g	ptl2 <sub>(8)</sub>	0xF + parent link <sub>(23)</sub>			$w_4$
up link <sub>(25)</sub>			ro2 <sub>(7)</sub>		$w_5$
ptl3 <sub>(8)</sub>		f	0xF + down link <sub>(23)</sub>		$w_6$
ptl4 <sub>(4)</sub>	l	sibling link <sub>(27)</sub>			$w_7$

## A.2 Preemption token

The preemption token is the list token of the *token-based-preempted-thread-list*:

map link <sub>(29)</sub>		cnt2	$w_0$
prev link <sub>(28)</sub>		cnt3 <sub>(4)</sub>	$w_1$
next link <sub>(28)</sub>		cnt4 <sub>(4)</sub>	$w_2$
lock link <sub>(20)</sub>	cnt1 <sub>(12)</sub>		$w_3$

**map link:** Pointer to the mapnode. Only the head of the list points to the mapnode.

**prev link:** Previous element in the list.

**next link:** Next element in the list.

**lock link:** The frame address to be locked.

**cnt:** Reference count. The counter is split into 4 parts  $cnt_1$  -  $cnt_4$ .

## A.3 Root overrun token

count <sub>(28)</sub>	ai <sub>(3)</sub>	t	$w_0$
-----------------------	-------------------	---	-------

**count:** Reference count. Note only threads of the same address space may reference this token.

**ai:** Reference cache. This cache contains a copy of the reference cache in the root mapnode, if this root mapnode has been overrun.

**t:** A root overrun occurred if this flag is set.

## A.4 Multiple pagesize support

The **dual node** data-structure stores the *dual nodes*. Those nodes are used to link the root array structures into the tree. Root array nodes are represented by the two data structures **rootnode** and **mid-rootnode**. The first is allocated only for the smallest hardware page-size. The second is allocated for all other page-sizes.

### A.4.1 Dual node

map link <sub>(29)</sub>	(2)	m	$w_0$
unmap link <sub>(29)</sub>	(2)	u	$w_1$
root link <sub>(30)</sub>	(2)		$w_2$
splitup list <sub>(28)</sub>	P (4)		$w_3$

**map link:** Pointer to the previous mapnode. This node has split mappings that are stored in the root array structure this node links to.

**unmap link:** Pointer to the next mapnode.

**m:** Always 0 (see mapnode).

**u:** Always 0 (see mapnode).

**root link:** Pointer to the root array structure.

**ps:** The pages of the mapping nodes in the subtree, the root array nodes point to are of the size ps.

**splitup list:** Pointer to the *split-token-based-preempted-thread-list*.

### A.4.2 Rootnode

*LL:*

map link <sub>(28)</sub>	$p_h$	i	m	$w_0$
--------------------------	-------	---	---	-------

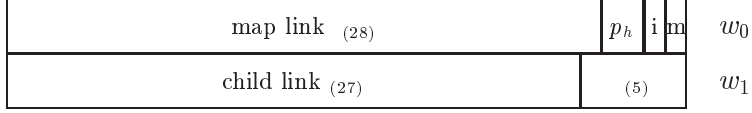
**map link:** Pointer to the next mapnode.

**m:** Set if the map link points to a dual node.

**i:** Mid = 0 because rootnode is allocated for smallest page-sizes only.

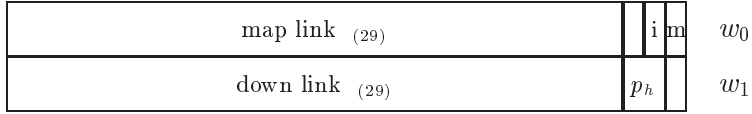
**p:** See mid rootnode below.

*LL-Tree:*



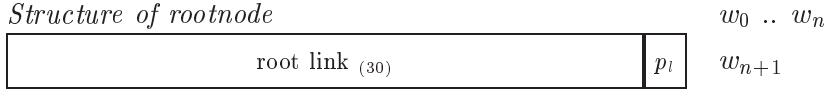
**child link:** Pointer to the left most child.

*LL-O1:*



**down link:** Pointer to the first leaf of the subtree.

#### A.4.3 Mid-Rootnode



**root link:** Pointer to the next root array.

**i:** Mid = 1, the root array node is a mid-rootnode

**p:** The pages of the mapping nodes in the subtree, the root array nodes point to are of the size p.

# Bibliography

- [ABB<sup>+</sup>86] M. J. Accetta, R. V Baron, W. Bolosky, D. B. Golub, R. F. Rashid, A. Tevanian, and M. W. Young. Mach: A new kernel foundation for UNIX development. In *Usenix Summer Conference*, pages 93–113, Atlanta, GA, June 1986.
- [AH99] A. Au and G. Heiser. *L4 User Manual Version 1.14*. Univerity of New South Wales, 1999.
- [AL91] A. W. Appel and K. Li. Virtual memory primitives for user programs. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, volume 26-4, pages 96–107, New York, NY, 1991. ACM Press.
- [And90] T. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.
- [ARJ97] J. Anderson, S. Ramamurthy, and R. Jain. Implementing wait-free objects on priority-based systems. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 229 – 280, Santa Barbara CA, 1997.
- [ARS89] V. Abrossimov, M. Rozier, and M. Shapiro. Generic virtual memory management for operating system kernels. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles (SOSP)*, volume 23-5, pages 123–136, 1989.
- [BCE<sup>+</sup>94] B. N. Bershad, C. Chambers, S. J. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage, and E. Gun Sirer. SPIN - an extensible micro-kernel for application-specific operating system services. In *ACM SIGOPS European Workshop*, pages 68–71, 1994.
- [BJ81] O. Babaoglu and W. Joy. Converting a swap-based system to do paging in an architecture lacking page-reference bits. In *8th Symposium on Operating Systems Principles (SOSP)*, pages 78–86, 1981.

- [BRE92] B. Bershad, D. Redell, and J. Ellis. Fast mutual exclusion for uniprocessors. In *5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Boston, MA, October 1992.
- [BSP<sup>+</sup>95] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the SPIN operating system. In *15th Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain, Colorado, 1995.
- [CD94] D. R. Cheriton and K. J. Duda. A caching model of operating system kernel functionality. In *Operating Systems Design and Implementation*, pages 179–193, 1994.
- [CHP71] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with readers and writers. *Communications of the ACM*, 14(10):667–668, 1971.
- [DSU] U. Dannowski, E. Skoglund, and V. Uhlig. *L4Ka Design Manual*. Universität Karlsruhe.
- [EGK95] D. R. Engler, S. K. Gupta, and M. F. Kaashoek. AVM: Application-level virtual memory. In *Fifth Workshop on Hot Topics in Operating Systems (HOTOS'95)*, pages 72–77, Orcas Island, WA, May 1995.
- [EGM<sup>+</sup>94] Y. Endo, J. Gwertzman, M. Seltzer, C. Small, K. A. Small, and D. Tang. VINO: the 1994 fall harvest. Technical Report TR-34-94, Harvard University, December 1994.
- [Elb] A. Elbs. *The Nutcracker Framework Description*. Universität Karlsruhe.
- [Elp99] K. Elphinstone. *Virtual Memory in a 64-bit microkernel*. PhD thesis, University of New South Wales, August 1999.
- [Fu97] S. Fu. A circular list-based mutual exclusion scheme for large shared-memory multiprocessors. *IEEE Transaction on Parallel and Distributed Systems*, 8(6):628–639, June 1997.
- [GC96] M. Greenwald and D. R. Cheriton. The synergy between non-blocking synchronization and operating system structure. In *Operating Systems Design and Implementation*, pages 123–136, 1996.
- [Hae01] A. Haeberlen. User level management of L4 kernel memory. In *2nd L4 Implementors Workshop*, 2001.
- [Han99] S. Hand. Self-paging in the Nemesis operating system. In *3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, New Orleans, Louisiana, February 1999.

- [HC92] K. Harty and D. R. Cheriton. Application-controlled physical memory using external page-cache management. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, volume 27 9, pages 187–197, New York, NY, 1992. ACM Press.
- [Hei01] G. Heiser. *Inside L4/Mips*. University of New South Wales, January 2001.
- [Her91] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.
- [HH01] M. Hohmuth and H. Härtig. Pragmatic nonblocking synchronization for real-time systems. In *USENIX Annual Technical Conference (USENIX '01)*, 2001.
- [Hil92] D. Hildebrand. An architectural overview of QNX. In *1st Usenix Workshop on Micro-kernels and Other Kernel Architectures*, pages 113–126, Seattle, WA, April 1992.
- [Hoh98] M. Hohmuth. The Fiasco kernel: Requirements definition. Technical Report TUD-FI-12, TU Dresden, December 1998.
- [HWL96] H. Härtig, J. Wolter, and J. Liedtke. Flexible-sized page-objects. In *5th International Workshop on Object Orientation in Operating Systems (IWOOOS)*, Seattle, WA, oct 1996.
- [Int02] Intel. *IA-32 Intel Architecture: Software Developer's Manual Volume 3*, 2002. Order Number: 245472-007.
- [JM98] B. Jacob and T. Mudge. Virtual memory: Issues of implementation. *IEEE Computer*, 31-6:33–43, June 1998.
- [KN93] Y. A. Khalidi and M. N. Nelson. The Spring Virtual Memory System. Technical Report SMLI TR-93-09, Sun Microsystems Laboratories, Mountain View CA (USA), 1993.
- [Knu97] D. Knuth. *The Art of Computer Programming: Fundamental Algorithms*. Addison - wesley, 3rd edition, August 1997.
- [KWS97] L. Kontothanassis, R. Wisniewski, and M. Scott. Scheduler-conscious synchronization. *ACM Transaction on Computer Systems*, 15(1):3–40, February 1997.
- [LA94] B. Lim and A. Agarwal. Reactive synchronization algorithms for multiprocessors. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*,, pages 25–35, October 1994.

- [LE96] J. Liedtke and K. Elphinstone. Guarded page tables on Mips R4600 or an exercise in architecture-dependent micro optimization. *Operating Systems Review*, 30(1):4–15, January 1996.
- [Lie94] J. Liedtke. Address space sparsity and fine granularity. In *6th SIGOPS European Workshop*, pages 78–81, Schloß Dagstuhl, Germany, September 1994.
- [Lie95] J. Liedtke. On  $\mu$ -kernel construction. In *15th ACM Symposium on Operating System Principles (SOSP)*, pages 237–250, Copper Mountain Resort, CO, December 1995.
- [Lie96] J. Liedtke.  $\mu$ -kernels must and can be small. In *5th International Workshop on Object Orientation in Operating Systems (IWOOS)*, pages 152–155, Seattle, WA, October 1996.
- [Map92] G. Mapp. *An Object-Oriented Approach to Virtual Memory Management*. PhD thesis, University of Cambridge Computer Laboratory, Januar 1992.
- [MCS91a] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1), February 1991.
- [MCS91b] J. M. Mellor-Crummey and M. L. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. In *Proc. Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1991.
- [MDP96] D. Mosberger, P. Drushel, and L. Peterson. Implementing atomic sequences on uniprocessors using rollforward. *Software Practice and Experience*, 26 1:1–23, 1996.
- [MS96] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 267–275. ACM Press, 1996.
- [Ros89] B. Rosenburg. Low-synchronization translation lookaside buffer consistency in large-scale shared-memory multiprocessors. In *12th ACM Symposium on Operating System Principles (SOSP)*, pages 137–146, Lichfield Park, AR, December 1989.
- [RTY<sup>+</sup>87] R. Rashid, A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, and J. Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. In *2nd Symposium on Architectural Support for Programming Languages and Operating Systems*, October 1987.

- [SRL90] L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. on Computers*, 39(9):1175–1184, September 1990.
- [Sto81] M. Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7):412–418, 1981.
- [Szm99] C. Szmajda. A new virtual memory implementation for L4 / MIPS. Undergraduate thesis, University of New South Wales, November 1999.
- [Tea] L4Ka Team. *L4 eXperimental Kernel Reference Manual Version X.2*. Universität Karlsruhe.
- [Tel91] P. J. Teller. *Translation-Lookaside Buffer Consistency in Highly-Parallel Shared-Memory Multiprocessors*. PhD thesis, New York university, 1991.
- [TKHP92] M. Talluri, S. I. Kong, M. D. Hill, and D. A. Patterson. Tradeoffs in supporting two page sizes. In *ISCA*, pages 415–424, 1992.
- [Uh198] V. Uhlig. Speicherverwaltung für den L4-Alpha Kern mit Echtzeitanforderungen. Undergraduate thesis, Technische Universität Dresden, 1998.
- [Uni00] Universität Karlsruhe. *Lecture notes: Microkernel Construction Course*, 2000. <http://i30www.ira.uka.de/teaching/coursedocuments/30/mkc-10e.pdf>, Slide 15ff.
- [Val95] J. D. Valois. *Lock-Free Data Structures*. PhD thesis, Faculty of Rensselaer Polytechnic Institute, Troy, New York, May 1995.