Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master's Thesis Nr. 47

Systems Group, Department of Computer Science, ETH Zurich

Virtual Memory in a Multikernel

by

Simon Gerber

Supervised by

Prof. Timothy Roscoe

November 2011 – May 2012

Informatik
Computer Science

Even though virtual memory is largely understood, and implementations perform well, in a single core environment, with the advent of multicore processors we have a new set of circumstances that the virtual memory system has to adapt to. As the trend of putting more and more cores on a single chip is continuing, the true-and-tried single-core virtual memory architecture does not scale any more in scenarios that require shared memory regions.

We try to address the problem of sharing memory regions on multicore systems by exploring how to design virtual memory in a “multikernel” – an approach to building an operating system where the set of cores is treated as a distributed system – where a kernel runs on each core. In particular, this thesis focuses on “Barrelfish”, a multikernel implementation that uses capabilities for managing its physical resources and for privilege control.

We look at the design of the virtual memory system in a multikernel environment from the bottom up, and present a new kernel interface that relies on capabilities as identifiers of mapped memory regions. As capabilities can be copied around freely, we can use the existing capability transfer mechanisms to set up shared memory regions in an efficient manner.

Overall, the results using the new kernel interface for mapping and unmapping regions are promising, showing performance improvements in orders of magnitude. These results show that the design is sound even though the unmap implementation does not perform as well as we could hope for small memory regions.

Acknowledgements

I would like to thank Prof. Timothy Roscoe for allowing me the opportunity to work on Barrelfish and for the valuable discussions and insights, Mark Nevill for letting me bounce ideas off him and for providing debugging help, Kornilios Kourtis, Tim Harris, Kevin Elphinstone, and Andrew Baumann for their input and answers to my questions, as well as many others in the ETH Zürich Systems Group for their assistance and feedback.
Additionally, I would like to thank my parents for all the support over the years, for letting me pursue my interests, as well as providing respite on many week-ends, and my sister and other friends for providing time away from work.

Contents

1 Memory Management in a Modern Operating System **3**
1.1 Motivation 3
1.2 Existing Memory Management Systems 4
1.3 Memory Management in a Multikernel 6

2 Barrelfish: A Multikernel Implementation **9**
2.1 Capabilities 9
2.2 Hardware abstraction in Barrelfish 10
2.3 The memory subsystem in libbarrelfish 11

3 Stateful Memory Management in Barrelfish **13**
3.1 Necessary Capability Functionality 13
3.2 Kernel Programming Interface 18
3.3 Libbarrelfish Adaptations 27
3.4 Memory Object Sharing 30

4 Performance **33**
4.1 Methodology 33
4.2 Expectations 33
4.3 Results 34
4.4 Analysis of our Methods 44

5 Conclusion **47**
5.1 Future Work 47

A X86 Hardware Support for Memory Management and Address Translation **49**
A.1 Segmentation 49
A.2 Paging 50
A.3 Privilege Levels 52

Bibliography **55**

1 Memory Management in a Modern Operating System

1.1 Motivation

A modern operating system (OS) has a couple of purposes: it abstracts the hardware from applications, it provides process separation and multitasking, and it manages physical resources (network connection, disk drives, physical memory). In this thesis, the focus lies on the management of physical memory in a multi-core system.

Two classic design points for operating systems are monolithic kernels and microkernels. A *monolithic kernel* is just what its name implies: a large, privileged application handling everything related to hardware, process management and process interaction, running in supervisor mode (i.e. that process can do everything that is possible given the hardware). Conversely, a *microkernel* tries to make the component that is running in supervisor mode as small as possible. This is often accomplished by having the kernel handling message passing and not much else. The “classic” operating system tasks are then handled by different servers (a memory server, a file system server, a network server, etc.)

However, seeing as today’s hardware gains more and more physical processing units, the classic approach of having one kernel that runs on all physical cores of a system is becoming unwieldy. This is largely because running the same kernel on all cores leads to (potentially) unnecessary sharing of kernel data structures. This also means that access to those structures must be mediated using locks or more advanced mechanisms for mutual exclusion such as read-copy-update. This is especially true for the memory management subsystem, where potentially many applications want to change the same data structure (the page tables) and it is essential that these operations succeed or fail quickly.

Recent efforts by Baumann et al. [1] show a new approach, called “multikernel” that tries to address these problems by exploring a new point in the operating system design space. Their approach has the advantage over a classic monolithic or even a microkernel in that it treats a multi-core system as if it were a distributed system, running a privileged CPU driver on each physical core. On top of the privileged CPU driver a trusted process – called the “monitor” – is running on each core. All monitors in the system cooperate in order to keep identical replicas of the OS state on each core and applications that need kernel services talk to their local monitor which then forwards requests to the appropriate core.

Regarding memory management, the advantages of this approach are readily visible: an application that wants to map a page in its virtual address space talks to the local monitor and CPU driver. The CPU driver then inserts the mapping in the hardware page tables. As page tables (like all other OS state) are replicated on all cores, this operation does not conflict with mapping requests of other applications running on other cores in the system.

Any operating system has to provide isolation of user applications from each other. Generally this is done by having a separate address space for each application and a trusted central service that tracks which application has permissions to access what resources (e.g. the Windows NT "Object Manager", cf. Windows Internals, Section 3.2 [15]). This central resource management service is a hindrance in harnessing the performance improvements that multiple cores provide, as it represents a serialization point for all concurrent kernel operations by running applications. However another decentralized and well understood way of granting resources to applications are capabilities [7, 9]. Capabilities are effectively keys to hardware resources and cannot be forged. This also means that once a process holds a capability it is sufficient to verify that the capability grants the appropriate rights before performing the requested action. This verification can be done by the kernel and does not need a trusted resource management service.

1.2 Existing Memory Management Systems

There are a number of existing memory management systems that do not closely follow the true and tried model of only having the kernel setting up translations from virtual to physical addresses.

An older design based on ideas of letting user applications act as memory managers, is demonstrated by the Mach microkernel [14]. An important feature of Mach's memory management system is the ability to handle page faults outside the kernel. This is accomplished by associating all memory objects (a memory object is a byte-indexed array which supports various operations such as read and write) with a task managing that memory object (this managing task is often called a "pager"). A pager in Mach has a "port" (that is, a connection) to the kernel over which the kernel can either request data from, or send notifications to the pager. By utilizing the pager interfaces in the Mach kernel, an application can provide its own pager (an "external" pager) if it wants to. To make application development easier Mach also provides a default pager (the "internal" pager) that manages all memory objects that have no associated pager.

Another approach that allows applications that cooperate to provide mappings to each other is shown in L4 [10, pp.8–9]. L4's memory management system is built around address spaces that contain all the data that is directly accessible to an application (a "thread" in L4). L4 allows these address spaces to be constructed recursively: an application can map parts of its address space into another application's address space (provided that the receiver accepts the mapping). Shared mappings can be revoked at any time by the application that installed the mapping and therefore the initiator of the sharing (the “mapper”) retains full control over its memory.

L4 also has a second way for applications to manage their address spaces. An application can “grant” a part of its address space to another application. In this case the initiator of the action (the “granter”) does not retain any control over the granted region and cannot revoke the grant and the receiver (the “grantee”) gains full control over the granted region.

To make mapping and granting efficient, L4 implements both operations on page tables, without copying any actual data.

Cheriton and Duda [3] who treat the kernel as a caching hardware abstraction layer instead of a service provider and Engler et al. [5] who first introduced the *exokernel* operating system architecture, where the kernel exports all hardware interfaces in a secure manner to untrusted operating systems are the first proponents of a OS model where the kernel does even less work than in a microkernel system, and where the basic operating system services are provided as libraries that each user program can link against. This approach, a minimal kernel and a library providing all the services, is therefore often called *library OS*.

The fact that these libraries are running in user space gives rise to the concept of *self-paging*. Self-paging was originally presented by Hand [6] as a quality of service mechanism for the Nemesis operating system. The idea behind self-paging is that each application is responsible for all of its allocations. This includes kernel structures associated with an application, e.g. page tables. The benefit of this approach is that each allocation can be accounted to some application’s memory budget. By carefully setting up memory budgets for different applications it is possible to give guarantees in regard to available memory to applications that need such guarantees. Also, with self-paging, handling a page fault can be accounted to the application’s CPU time.

A recent effort on library OSes by Porter et al. [13], shows a Windows 7-based library OS that is able to run commercial releases of various Microsoft applications, including Microsoft Excel and PowerPoint. They show that a small number of abstractions (namely threads, virtual memory and I/O streams) are sufficient to run a library OS version of Windows 7.

Even though they do not focus on user level memory management in the “Corey” operating system, Boyd-Wickizer et al. [2] show an approach for multi-core memory management that looks similar to the recursive mappings of L4. They have “address ranges” that are very similar to L4’s mappings. In Corey, these address ranges are a kernel abstraction and can be shared between applications and can be mapped in an address space as if they were normal pages, creating way to setup an address space that contains shared and private mappings. However, Corey is lacking a sound model of how such shared address ranges can be revoked.

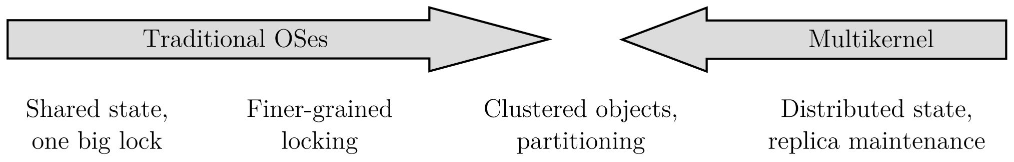
Traditional OSes
Multikernel
Shared state, one big lock
Finer-grained locking
Clustered objects, partitioning
Distributed state, replica maintenance

Figure 1.1: Sharing granularity (from [1])

1.3 Memory Management in a Multikernel

As shown in Figure 1.1 there are different levels of sharing and locking granularity in the operating system design space. Traditional OSes like Windows and Linux started out on the far left[1], but are adapting to multi-core environments by moving to finer-grained locks and starting to replicate shared state in order to optimize concurrent accesses on different cores. On the other hand, the multikernel approach is situated at the far right of the spectrum, and replicates all OS state. This has multiple benefits in a multi-core environment:

- It is possible to apply distributed systems techniques to ensure that the state replicas are in sync and sharing can be treated as an optimization for replication if the hardware facilitates sharing by providing cache-coherence.
- Inter-core communication is made explicit and can thus be optimized for the hardware environment.
- The OS structure is made hardware-neutral, to the point of being able to have a single operating system running on cores with different architectures [11].

Given a multikernel, there are a few possibilities to consider when wanting to provide a shared memory model to user applications. This mainly relates to how or if page tables are shared. First off, you can replicate all page tables, meaning that you then share individual pages. Second, you can share the parts of the page table structure that contain the shared mappings. Third and last, you can share all page tables in the case of wanting to provide an application a classic multi-threaded single address space environment.

Also, as the functionality provided by the kernel and monitor is minimal, a library OS on top of a multikernel makes application development much easier. Seeing that a multikernel most likely provides a library OS implementation, we need to consider two interfaces for managing memory:

- A kernel interface that allows the modification of the hardware page tables
- A library interface that provides a higher level abstraction to manage memory to the user

In the rest of this thesis, we introduce “Barrelfish”, a multikernel implementation in Chapter 2, then show a new kernel interface for Barrelfish and the prerequisites for that

[1]An example: the “big kernel lock” in Linux prior to version 2.6.39

interface as well as a sketch of higher level memory object management in Chapter 3. Finally, in Chapter 4 we compare the performance of the new kernel interface to the previous implementation in Barrelfish and discuss the improvements in the new design.

2 Barrelfish: A Multikernel Implementation

Barrelfish is a research implementation of the multikernel model as presented by Baumann et al. [1].

Barrelfish manages resources using *capabilities*, and has a relatively feature-rich library OS accompanying a small trusted kernel and a trusted inter-core synchronization server called "monitor". User applications on Barrelfish are called "domains" and have one "dispatcher" per core they run on. The unit of scheduling is therefore a dispatcher, and scheduling is done per core. Barrelfish provides user applications a default set of classic OS services in the `libbarrelfish` library.

Additionally, Barrelfish has a no-allocation policy in the kernel. This means that the monitor has to allocate all memory the kernel needs. The benefits of this policy are that all memory allocations can be accounted to a specific domain, and it is possible to put hard limits on the memory consumption of domains. On the other hand, the downside of the policy are the complications that arise when having to make sure that a kernel operation always has enough free memory to complete.

2.1 Capabilities

2.1.1 Capability types

Barrelfish's capabilities, which are modeled after seL4's capabilities [9], are typed. This means that there are different capability types for different resources and these types are organized in multiple type trees. An example: The capability tree representing physical memory has PhysAddr (Physical Address Range) as root. PhysAddr capabilities can then be retyped to either RAM (zeroed memory) or DevFrames (non-zeroed memory, necessary for BIOS regions, memory mapped device registers and similar regions). RAM capabilities can then in turn be retyped to CNode (used for storing capabilities), VNode (capabilities for page table frames), Dispatcher (identifying a process), and Frame capabilities. Currently all capabilities derived from PhysAddr have sizes that are a power of two. There are also a couple other capabilities such as a Kernel capability (which grants the monitor special privileges), capabilities identifying legacy IO ranges (IO capabilities) and various others. Additionally most kernel operations in Barrelfish (e.g. mapping and unmapping pages) are expressed as capability invocations. Capability invocations are system calls (syscalls) that take a capability as first argument and an operation on that capability as second argument.

2.1.2 Capability Storage

All capabilities are stored in CNodes. Each domain has a root CNode that is allocated when the domain is started. CNodes are special regions of memory that only the kernel and monitor (through the kernel) can access, and contain an array of capabilities. As CNodes themselves also have an identifying capability, all CNodes for a domain form a tree. Thus each capability can be identified by its address in the CNode tree.

2.1.3 Mapping Database (MDB)

In addition to being stored in a CNode, all capabilities are also inserted in a doubly linked list called the mapping database (MDB). The MDB allows looking up a capability's children, copies and ancestor. This functionality is needed to make capability operations like retype work (e.g. retype only succeeds if the capability that is being retyped has no children). However some operations on the MDB are not very efficient due to the representation of a tree in a linked list.

2.1.4 User space representation and Invocation

User space has `caprefs` that represent a CNode slot. A user application can use `caprefs` in order to perform any capability operations on the capability stored in the CNode slot represented by the `capref`. In order to perform a capability operation user space has to do a capability invocation. For complex capability operations such as retyping or copying a capability, the invocation is done on a CNode (typically the root CNode of the domain). For other operations, such as unmapping a Frame capability from a page table or getting the physical base address and size of a Frame, the invocation is done directly on the capability representing the page table or Frame respectively.

Even though user applications can create any `capref` they desire, as the capability itself never leaves trusted code (kernel and monitor) and all capability operations check capability rights, user applications cannot maliciously acquire rights to capabilities.

Capability invocations have the general form `syscall(SYS_INVOKE, capref, command, ...)` and the kernel will always verify that the domain performing the invocation has the necessary rights on all supplied capabilities.

2.2 Hardware abstraction in Barrelfish

On x86, Barrelfish uses a protected flat segmentation model (cf. Section A.1 for details on x86 segmentation) that contains a code and data segment for both user level and kernel code. All segments start at address zero and the kernel segments have privilege level zero, while the user segments have privilege level three (cf. Section A.3).

On x86, in addition to segmentation (which is mandatory), Barrelfish uses the paging hardware to provide classic virtual to physical address translation. Also, the different page table types (see Table A.1) are represented by capability types and the kernel enforces the layout of the page table tree (e.g. for x86, page directory entries have to be page tables).

In user space, Barrelfish borrows the BSD concept of keeping the hardware-dependent bits of memory management (allocating page tables, inserting mappings, and keeping track of the hardware-specific page table tree) in a physical map (pmap) and a more general interface built on virtual memory regions (vregions) and virtual address spaces (vspaces). Having this platform-independent API keeps user level code portable. Additionally, new hardware-specific optimizations benefit not only new applications but also existing ones.

If necessary, a user application can also access the pmap directly in order to implement customized memory management policies in order to improve its performance.

2.3 The memory subsystem in libbarrelfish

Barrelfish provides both malloc/free and an interface using memory objects (memobjs) and virtual regions (vregions). However most of the free functionality does not actually return allocated pages to the system, but rather just keeps the freed memory around for later allocation requests. Similarly, but more noticeably, most memobj types do not implement unmap. This leads to special-purpose memory pool implementations being scattered around the tree, especially when the user needs control over map flags.

The current implementation of virtual memory management in Barrelfish is (for lack of a better term) "map-only", meaning that most of the unmap functionality is missing. While there is a unmap system call and related kernel code, user space does not make much use of it other than when changing permissions, and the vspace/vregion interface supports unmapping regions only in a few cases. Additionally the current implementation is not very efficient as the state is mostly kept in linked lists.

At kernel level, mapping a page is treated as a special case of minting a new capability from the Frame capability that is being mapped. The idea behind this implementation of mapping a page is that it would make it easy to create shadow page tables that reflect which capabilities are mapped where (as there would actually be a copy of the mapped capability next to the page table entry). However, as Barrelfish's capabilities are 64 bytes large, the shadow page tables are not constructed and only the page table entry is set up.

Due to the shadow page tables only existing implicitly, it is not possible to undo mappings when a capability is revoked (as should be done) because we do not store the copies that are in the shadow page tables in the mapping data base (MDB).

3 Stateful Memory Management in Barrelfish

A robust memory management system is necessary for reliable and safe shared memory regions. Making a capability-based memory management system robust in a multi-core situation requires unmapping regions associated with capabilities that are being deleted or revoked.

This is not possible with the current implementation in Barrelfish's kernel, as the kernel does not know where a capability might be mapped. To enable this functionality we introduce the concept of a mapping at kernel level. These mappings are associated with a capability (or rather with one copy of a capability), and provide enough information to unmap a region whose associated capability is deleted. The downside of associating mappings with capabilities is that we need a copy of the capability for each mapping associated with that capability.

As the mappings stored in the capabilities are accessible only by the kernel, the system is secure as there is no way a malicious user application can trick the kernel into unmapping a region that it has no control over (holds the associated capability).

3.1 Necessary Capability Functionality

In order to be able to undo mappings given a capability to delete, we need to have a way to lookup those mappings. One possibility are shadow page tables containing a copy of the capability that is mapped to that entry. While this is possible with small capabilities, with larger capabilities (Barrelfish's capabilities are 64 bytes large) the cost of having a copy of the matching capability for each page table entry is large and we would like another, more space efficient way of getting the capability corresponding to a page table entry.

The solution we have come up with is to store the physical address of the first page table entry of the mapping in the capability as well as the number of pages belonging to the mapping and the offset in the physical range identified by the capability. However, storing three more 64 bit values in the capabilities means that we have to increase the capability size to 128 bytes. On the other hand, we only need one capability per mapping and thus the overhead of storing 128 byte sized capabilities is much smaller than for shadow page tables. The reason why the mapping state has to be inside the capabilities themselves is because capabilities are the only place in-kernel where state can be stored.

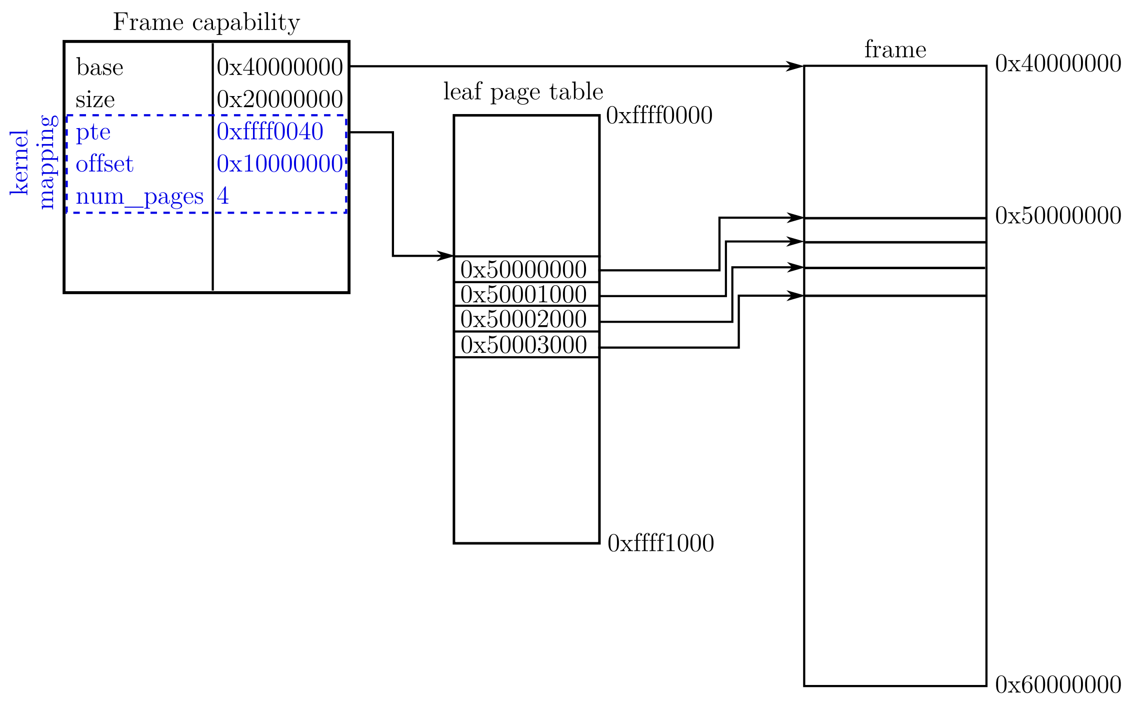

Figure 3.1: Kernel mapping data structure

Figure 3.1 shows an example of a Frame capability that represents the physical memory between addresses `0x40000000` and `0x60000000`, and that has a part mapped in a page table that is stored in physical memory at address `0xffff0000`. The mapping spans four 4 KiB pages, and starts at offset `0x10000000` (address `0x50000000`) of the physical memory identified by the frame capability. The fields that form the kernel mapping are highlighted in Figure 3.1

In order to provide selective translation lookaside buffer (TLB) shootdown which is necessary to guarantee that no process retains access to physical memory if its capability to that memory is revoked, we need a way to lookup a capability given its physical address. This can be achieved by having an ordering on capabilities and building a tree of all capabilities from that ordering.

3.1.1 Ordering

A sensible ordering on capabilities has to involve the capabilities' types, their type relationships, the addresses of the memory they identify (if applicable), and their size. Also, such an ordering should make it reasonably easy to find the copies, descendants or ancestor(s) of a given capability.

The ordering we want essentially partitions the capability set by type root, and then orders the capabilities first by ascending address and then by descending size. This means that we can compare two capabilities by comparing the tuple shown in Equation 3.1 for both of them.

$$(\mathit{type\_root}, \mathit{address}, -\mathit{size}, \mathit{type}) \tag{3.1}$$

As capability types can have additional fields that determine if two copies of a given capability type are equal, we have to modify the tuple for the ordering as shown in Equation 3.2.

$$(\mathit{type\_root}, \mathit{address}, -\mathit{size}, \mathit{type}, \text{additional equality fields}) \tag{3.2}$$

In order to distinguish two copies of the exact same capability, we also introduce a tiebreaker field in the ordering. We use the (local) memory address of the capability structure itself as tiebreaker as shown in Equation 3.3.

$$(\mathit{type\_root}, \mathit{address}, -\mathit{size}, \mathit{type}, \ldots, \mathit{capaddr}) \tag{3.3}$$

Having defined that ordering, we need to implement a comparison function that actually uses the ordering. As Barrelfish has a domain specific language (DSL) for its capability types (Hamlet), implementing the comparison function meant modifying Hamlet's DSL compiler. To make the implementing the comparison function easier, we also added functions that return the address and size of a capability to Hamlet.

3.1.2 Index

In order to have lookups based on that ordering which are faster than $O(n)$ for n the number of capabilities, we need to have a suitable index structure that fits into the capability structures, does not need in-kernel allocation and ideally also supports range queries for the use case of looking up capabilities by address.

We get fast lookup by ordering from any binary search tree. Mark Nevill [12] implemented an AA-tree as a replacement for the linked list representation of a binary search tree. However, to enable storing and querying intervals (which most capabilities are) we need either an interval tree or an augmented tree. As we already had an implementation of a binary tree using the *low* values of the intervals in the ordering keys, we chose to extend that implementation according to the description of an augmented tree by Cormen et al. [4, pp. 311–317]. This meant adding extra fields to the tree nodes that represent the maximum ending address of the capabilities in both subtrees and the node itself as well as the `type_root` of the capability determining the *high* value. We need the `type_root` of the *high* value because we have only one index tree for all type trees. These attributes can be maintained in $O(h)$ steps during each addition and removal of a node, h being the height of the node that is added or removed. Also, rotations while inserting or deleting need to update these fields in the affected nodes.

3.1.3 Range Query

Given a two intervals A and B, $A.low \leq B.low$ without loss of generality, and an augmented tree containing intervals, we now know that they overlap iff

$$A.low \leq B.high \wedge A.high \geq B.low$$

When searching for an interval I in the tree, we can therefore disregard all nodes to the right of a node N with $N.low > I.high$ and all nodes N with maximum ending address $N.end < I.low$. Given the additional restrictions on capabilities, the only relationships between ranges are outlined in Figure 3.2, we can even do different types of range query which will return different capabilities in relation to the queried range R. These types are listed below (see Figure 3.2 for a visual representation of the different result types).

surrounding if we specify we want a surrounding result, the result of the query (if any) will be a capability C that is at least as large as the queried interval but not larger than any other capability that is larger than the queried interval, i.e. that satisfies

$$\begin{aligned} C \neq 0 \Rightarrow &(C.low \leq R.low \wedge C.high \geq R.high) \wedge \\ &(\forall c \in X : C.high - C.low \leq c.high - c.low) \\ \text{where} & \\ &X \text{ is the set of all capabilities containing } R \end{aligned}$$

inner An inner result is a capability C that is inside the queried range and is at least as large as any other capability inside the range, i.e. which satisfies

$$\begin{aligned} C \neq 0 \Rightarrow &((C.low \geq R.low \wedge C.high < R.high) \vee \\ &(C.low > R.low \wedge C.high \leq R.high)) \wedge \\ &(\forall c \in X : C.high - C.low > c.high - c.low) \\ \text{where} & \\ &X \text{ is the set of all capabilities contained in } R \end{aligned}$$

partial A partial result is a capability C that overlaps one end of the queried range, but not the other, i.e. which satisfies

$$\begin{aligned} C \neq 0 \Rightarrow &((C.low > R.low \wedge C.low < R.high \wedge C.high > R.high) \vee \\ &(C.high > R.low \wedge C.high < R.high \wedge C.low < R.low)) \end{aligned}$$

3.1.4 Capability locality

To make cross-core capability operations efficient, all capabilities need an owner core. All capability operations then have to go through that owner core, so that all invariants can

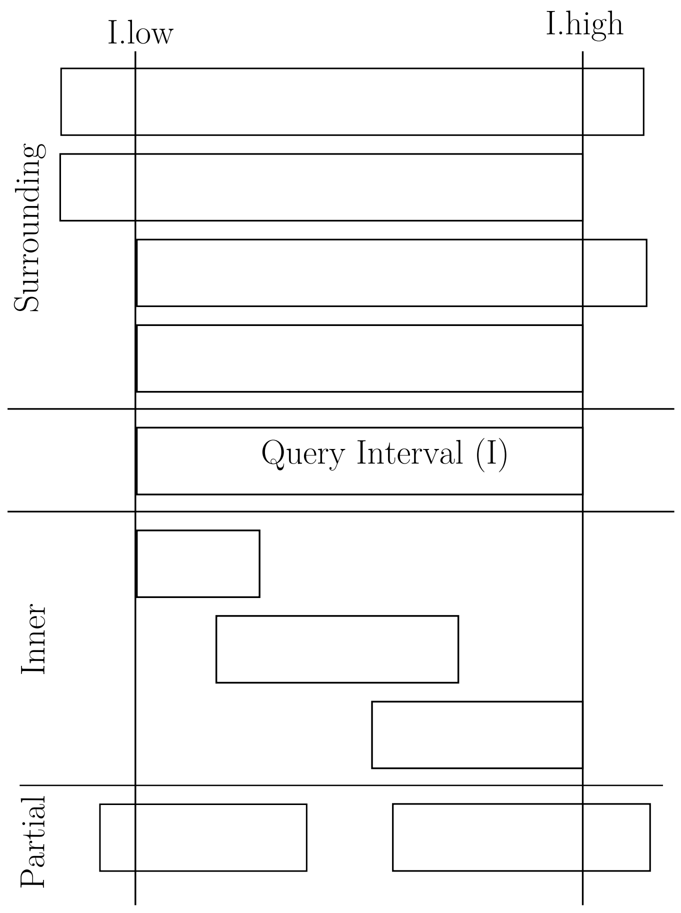

Figure 3.2: Capability overlap possibilities

be kept. There are some operations that are not impacted by the fact that each capability now has an owner core. As capabilities have an owner core, it has to be possible to move a capability from one core to another. Thus a capability system that supports owner cores has to have a move capability operation.

3.1.5 Remote invocation

If we want to be able to share page tables in such a way that all sharing participants can install new mappings in those shared page tables we need to have a way to handle invocations on non-local capabilities, as the new system allows non-local capabilities of any type. If we want to have remote invocations that do not need a broadcast in order to find the owner core of the non-local capability, we need to have a way to store the owner core of a non-local capability in the capability itself or next to it.

We also need to have a way to update all these references to the owner core if the capability is moved to another core.

One possibility is to eagerly update all references to the owner core if a capability is moved. The downside of this approach is that a move operation can get very expensive as it has to notify all cores with non-local copies that they have to change their information to reflect the new owner core.

Another possibility is to leave a marker on the old owner core that redirects remote invocations to the new owner core and also notifies the initiator of the remote invocation to update its owner field. While this approach results in a cheaper move operation, remote invocations are potentially slower as they might have to chase a chain of such redirections if the capability was moved multiple times.

3.2 Kernel Programming Interface

When designing an interface we need to know what arguments we need for the operation to which we design the interface. The absolutely necessary arguments for a map call are

- The virtual address at which to map
- A reference to a physical memory region to map
- An offset into the physical memory region
- A mapping size

Usually a map call also takes mapping flags as an argument in order to provide the user control over the page permissions of the mapping.

Looking at that argument list we see that the virtual address can be represented in two fundamentally different ways that will influence the whole memory subsystem. The first way to represent the address is to use the virtual address itself and the address space identifier belonging to the address. The second representation is the tuple $(ptable, entry)$ describing the page table entry that corresponds to the full virtual address in the given address space.

When using the first representation (the virtual address itself) we run into problems in a fully self-paged system (as Barrelfish is), because we might have to abort a kernel operation (mapping a page) when a non-root page table is not mapped yet. We run into problems in that case because user space is responsible for mapping any memory it needs, including the memory that is used as page tables. This also makes the response time for mapping a region very unpredictable, as there might be multiple attempts at mapping that are aborted because of missing page tables (assuming the user space implementation does not ensure that all page tables are available beforehand).

However, if we use the $(ptable, entry)$ representation for virtual addresses, we never have that particular problem. That is, in order to map a page, we provide a reference to the page table in which we want to change a specific entry. The kernel then does not care if that page table is installed yet or not, only that it exists. Using this approach we also gain the flexibility we need in order to construct partial page tables and then quickly update whole parts of the page table tree, e.g. by overwriting a leaf page table mapping with the address of a superpage[1].

With the $(ptable, entry)$ representation we additionally have a superpage-agnostic interface for free, as mapping a superpage should be identical to mapping a normal page, only in a different level of the page table hierarchy.

Another consideration is the need to flush the translation lookaside buffer (TLB) when removing mappings. This is especially expensive if the mapping was shared across cores as we then have to request a remote TLB flush. The cost of these remote TLB flushes can be minimized by specifying the virtual address for which the TLB has to be invalidated.

As we need the virtual address for TLB invalidation, the interface that takes the full virtual address as argument makes selective TLB invalidation very easy. On the other hand, having the virtual address as $(ptable, entry)$, we need to walk up the page table tree, reconstructing the virtual address as we go along. This is quite expensive as it involves multiple range queries to find the page table entries in the higher level page tables and might lead to a poorly performing unmap operation.

Nevertheless, as we mostly only map memory on the fast path of applications, and unmap is more of a cleanup operation, we choose to implement the $(ptable, entry)$ interface. This gives us an easy design for mapping regions, seeing as all page tables need to be mapped by user space anyway. We also utilize the new capability index to provide fast virtual address reconstruction for selective TLB shootdown, making unmap perform reasonably well.

[1]that is, a page that is larger than the base page size (which is commonly 4 KiB)

3.2.1 Kernel Mapping Contents

In order to make that approach work – as we have to store the mapping meta data in constant space in the kernel – we need to restrict in-kernel mappings to a single leaf page table. Also, not doing so would mean that if we want to relocate whole page tables we would potentially destroy mappings that overlap multiple leaf page tables. If we constrain the kernel to single leaf page table mappings, we only need to store the physical address of the first page table entry of the mapping, the physical offset into the memory region identified by the capability we are mapping, and the number of pages we are mapping in order to implement all four operations described in the following sections.

3.2.2 Reconstructing the virtual address of a page table entry

In order to reconstruct the virtual address of a page table entry we need to walk up the page table tree and combine all entry indices (effectively performing the reverse operation of what you would do to figure out the page table entry in the first place). The problem with this approach is to find the page table entry for the page table we are looking at right now. However, if we already keep track of where a Frame capability is mapped, we can do the same with page table capabilities and then use a range query to get the capability for the address of the page table entry that is stored in the page table we are currently looking at.

Algorithm 1 Reconstruct a virtual address

```
function COMPILE_VADDR($ptable$: capability, $entry$: slot)
    $vaddr \leftarrow 0$
    $sw \leftarrow BASE\_PAGE\_BITS$                ▷ assumes that $ptable$ is a leaf page table
    $vaddr \leftarrow (entry \ \& \ PT\_MASK) \ll sw$
    $old \leftarrow ptable$
    repeat
        $sw \leftarrow sw + PT\_BITS$
        $next \leftarrow$ FIND_CAP_FOR_ADDRESS($old.pte$)
        $offset \leftarrow (old.pte - next.base) \ / \ PT\_ENTRY\_SIZE$
        $vaddr \leftarrow vaddr \mid ((offset \ \& \ PT\_MASK) \ll sw)$
        $old \leftarrow next$
    until found root page table
    $asid \leftarrow next.asid$                ▷ extract address space id from root page table
    return $asid, vaddr$
end function
```

Operators: '&' is bitwise AND. '|' is bitwise OR. '≪' shifts the left-hand side left by right-hand-side many bits.

As we need to be able to reconstruct virtual addresses for arbitrary page table types, we modify Algorithm 1 so that the initial shift width is calculated based on the type of

Example: Recreating a Virtual Address

This example shows the steps Algorithm 1 makes for IA-32 paging[1].
First, we use the page table entry we get as an argument to construct the first part of the virtual address as shown in Figure 3.3.

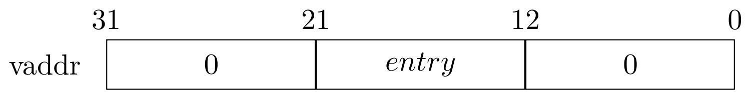

Figure 3.3: Partial virtual address after first step

In a next step, we use the page table capability we get as an argument to do a range query that should return the page directory in which the page table is installed. We then calculate the page directory entry by subtracting the base address of the page directory from the *pte* field of the page table capability.

$$pdir\_entry = \frac{ptable\_cap.pte - pdir\_cap.base}{PT\_ENTRY\_SIZE}$$

We then put the page directory entry we just calculated into the virtual address (see Figure 3.4).

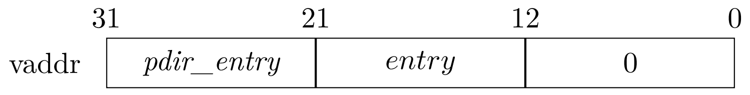

Figure 3.4: Reconstructed virtual address

If the page table hierarchy is larger, we would now repeat these steps until we find the root page table. After finding the root page table and adding the root page table entry to the reconstructed virtual address, we return that address.

[1] the IA-32 page table tree is built of a page directory and 1024 page tables, cf. Section A.2

ptable. Algorithm 2 shows how the initial shift width is computed in the implementation of the new kernel interface in Barrelfish.

Algorithm 2 Compute initial shift width for reconstructing a virtual address

```
// shift at least by BASE_PAGE_BITS for first vaddr part
size_t shift = BASE_PAGE_BITS;

// figure out how much we need to shift
//
// A couple of cases have fallthroughs in order to avoid having
// multiple calls to vnode_objbits with the same type argument.
switch (ptable->cap.type) {
    case ObjType_VNode_x86_64_pml4:
        shift += vnode_objbits(ObjType_VNode_x86_64_pdpt);
    case ObjType_VNode_x86_64_pdpt:
        shift += vnode_objbits(ObjType_VNode_x86_64_pdir);
    case ObjType_VNode_x86_64_pdir:
        shift += vnode_objbits(ObjType_VNode_x86_64_ptable);
    case ObjType_VNode_x86_64_ptable:
        break;

    case ObjType_VNode_x86_32_pdpt:
        shift += vnode_objbits(ObjType_VNode_x86_32_pdir);
    case ObjType_VNode_x86_32_pdir:
        shift += vnode_objbits(ObjType_VNode_x86_32_ptable);
    case ObjType_VNode_x86_32_ptable:
        break;

    case ObjType_VNode_ARM_l2:
        shift += vnode_objbits(ObjType_VNode_ARM_l1);
    case ObjType_VNode_ARM_l1:
        break;

    default:
        return SYS_ERR_VNODE_TYPE;
}
```

3.2.3 Mapping a region

The biggest changes to the map invocation are that it is no longer done on the root CNode, but on the VNode where the page(s) need to be mapped and that we specify the number of pages to map. The new algorithm for mapping a kernel region is shown in Algorithm 3

Algorithm 3 Mapping a region

```
function MAP($ptable$: the page table where this should be mapped,
             $cap$: a pmem capability to map,
             $entry$: the offset in $ptable$ at which to map,
             $cap\_offset$: the offset into the cap to map,
             $num\_pages$: the size of the new mapping)

    if $cap.pte \neq 0$ then                       ▷ Check that cap is not mapped already.
        return $ERR\_CAP\_ALREADY\_MAPPED$
    end if

    $last\_entry \leftarrow entry + num\_pages$
    if $last\_entry > PT\_SIZE$ then
        ▷ Check that the number of entries to map does not overlap a leaf page table
        return $ERR\_MAP\_SIZE$
    end if

    $as, vaddr \leftarrow$ COMPILE_VADDR($ptable$, $offset$)
    $paddr \leftarrow cap.base$
    $offset \leftarrow 0$

    $cap.pte \leftarrow$ GET_ADDRESS($ptable$) $+ entry$          ▷ Set mapping meta data
    $cap.mapped\_pages \leftarrow num\_pages$
    $cap.mapped\_offset \leftarrow cap\_offset$

    repeat                              ▷ Map cap in chunks of size BASE_PAGE_SIZE
        if HAS_MAPPING($ptable, entry$) then
                                        ▷ overwrite old mapping and flush TLB entries
            ZERO($ptable$, $entry$)
            FLUSH_TLB($as$, $vaddr + offset$)
        end if
        INSERT_MAPPING($ptable$, $cap$, $cap\_offset + offset$, $entry$)
        $entry \leftarrow entry + 1$
        $offset \leftarrow offset + BASE\_PAGE\_SIZE$
    until $offset \geq num\_pages * BASE\_PAGE\_SIZE$
end function
```

We can see that the actual operation in the kernel takes longer, as there is a tight loop that sets up the individual page mappings in the kernel itself. However, as the same loop is not necessary in user space anymore and the number of system calls is decreased (and therefore also the overhead of switching from and to kernel mode), the total response time on a map request should stay about the same for one page and get smaller for mapping multiple pages.

With some small adaptations we can use the algorithm presented in Algorithm 3 not only for mapping Frame capabilities but also for mapping VNode capabilities. The current implementation checks if the map request is valid, and then calls a map handler that matches the map request. Currently the only map requests that are handled are either mapping a Frame capability region using normal (4 KiB) pages, or mapping one page table frame (provided that the VNode type of *cap* matches the entry type of the VNode type of *ptable*).

3.2.4 Unmapping a region

When the kernel gets an unmap request, it will try to find a capability that has mapping meta data that matches the parameters in the unmap request (those parameters are again a leaf page table, a slot index in that page table, and a number of pages to unmap). It picks the first capability that matches the request (if such a capability exists) as the capability to unmap (cf. Algorithm 4).

Algorithm 4 Find capability for mapping

```
function LOOKUP_CAP_FOR_MAPPING(
                    paddr: the physical address to look up,
                    pte: the page table entry address of the mapping)

    cap ← FIND_CAP_FOR_ADDRESS(paddr)
    orig_cap ← cap
    repeat
        ▷ Loop through all copies of the capability identifying paddr
        ▷ until we either run out of copies or find one that is mapped
        ▷ at entry in ptable.
        if $cap.base + cap.mapped\_offset = paddr$ and $cap.pte = pte$ then
            ▷ Found a good copy
            return cap
        end if
        cap ← GET_NEXT(cap)                    ▷ Lookup next copy of cap in MDB
    until not CAP_IS_COPY(cap, orig_cap)
    return NULL
end function
```

After finding a matching capability, unmap proceeds to zero out the requested number of page table entries and clears the mapping meta data in the capability. Clearing the mapping data allows the user to remap the newly unmapped copy of the capability.

Algorithm 5 Unmap a region

```
function UNMAP(ptable: the page table in which to unmap,
               entry: the entry in the page table,
               num_pages: the number of pages to unmap)
    $as, vaddr \leftarrow$ COMPILE_VADDR($ptable, entry$)
    $paddr \leftarrow$ GET_PADDR($ptable, entry$)
    $pte \leftarrow$ GET_ADDRESS($ptable$) $+ entry$
    $cap \leftarrow$ LOOKUP_CAP_FOR_MAPPING($paddr, pte$)
    if $cap = NULL$ then
        return $ERR\_MAPPING\_NOT\_FOUND$
    end if
    if $cap.mapped\_pages \neq num\_pages$ then
        return $ERR\_MAP\_SIZE$
    end if
    $c \leftarrow 0$
    repeat
        ZERO($ptable$, $entry + c$)
        $c \leftarrow c + 1$
    until $c \geq num\_pages$
    FLUSH_TLB($as$, $vaddr$, $vaddr + num\_pages * BASE\_PAGE\_SIZE$)
    $cap.pte \leftarrow 0$
    $cap.mapped\_offset \leftarrow 0$
    $cap.mapped\_pages \leftarrow 0$
end function
```

After unmapping a region, the TLB entries covering that region are flushed. However, on x86 architectures, it is not always clear if the cost of doing a selective flush for each unmapped page is cheaper than the cost of reloading the TLB after reloading the `%cr3` register that holds the root page table address.

3.2.5 Deleting a capability

In order to guarantee that an application is not able to access memory after the associated capability is deleted, the capability deletion mechanism has unmap memory that is mapped. This can be done by doing a range query on the page table entry (if any) that is stored in the capability which yields the page table capability that is needed to perform an unmap. With the page table capability and the entry index we can do a normal unmap using the number of pages stored in the mapping meta data.

Algorithm 6 Unmap region associated with capability that is being deleted

```
function DELETE(cap: the pmem capability that is being deleted)
    if $cap.pte = 0$ then                                   ▷ Do nothing if cap is not mapped
        return $ERR\_OK$
    end if
    $ptable \leftarrow$ FIND_CAP_FOR_ADDRESS($cap.pte$)
    if $ptable = NULL$ then                                 ▷ Assume all OK if page table not found
        return $ERR\_OK$
    end if
    $entry \leftarrow cap.pte - ptable.base$
    $as, vaddr \leftarrow$ COMPILE_VADDR($ptable, entry$)
    $c \leftarrow 0$
    repeat          ▷ Remove all BASE_PAGE_SIZE sized mappings belonging to cap
        ZERO($ptable$, $entry + c$)
        FLUSH_TLB($as$, $vaddr + c * BASE\_PAGE\_SIZE$)
        $c \leftarrow c + 1$
    until $c \geq cap.mapped\_pages$
    return $ERR\_OK$
end function
```

3.2.6 Selective TLB shootdown

While it is easy to flush the whole TLB after unmapping a page, performance might be better if we selectively only flush the pages we actually changed. To do a selective TLB flush on x86 and ARM we need the virtual address of the TLB entry we want to invalidate. As we already have a way to reconstruct the virtual address (see Section 3.2.2), we only need to replace the TLB reloading (see Algorithm 7 for how it is done on x86) with the algorithm presented in Algorithm 8 (also x86 specific).

Algorithm 7 Reloading the whole TLB by re-setting the root page table address

```
mov %cr3, %rax
mov %rax, %cr3
```

Algorithm 8 Flushing selected addresses from the TLB

```
for (int i = 0; i < num_base_pages; i++) {
  genvaddr_t addr = vaddr + i * BASE_PAGE_SIZE;
  __asm__ __volatile__("invlpg %0" : : "m" (*(char*)addr));
}
```

3.2.7 Missing kernel interface functionality

There are a few parts a full-featured kernel interface for memory management would provide that we left out. The most prominent example is the modification of page permissions. Right now Barrelfish supports modifying page flags by unmapping and then remapping each page with the new set of flags. However, the current implementation shows inconsistent states to the user in a multi-threaded environment, as one thread might access a region whose flags are being modified by another thread at the same time (think garbage collection). One way to partially prevent this is to provide a “modify flags” function in the kernel interface.

Updating page permissions

This *modify flags* function has the kernel mapping and the new set of flags as arguments and changes the flags on all pages belonging to the mapping to the new set of flags. Given the data contained in a kernel mapping modifying the flags on n pages takes only $O(n)$ cycles.

3.3 Libbarrelfish Adaptations

Seeing as the kernel interface for mapping and unmapping pages has new semantics, it follows that the default self-paging implementation (in `libbarrelfish`) has to be adapted to the new design. First off, the structure of the user level representation of the page table tree was changed to allow nodes that represent a kernel mapping as described in Section 3.2.1. Also, in order to handle mappings that are comprised of multiple kernel mappings, libbarrelfish transparently creates and stores new copies of the capability that is being mapped (because each copy of a capability can be mapped only once) – the new map algorithm is shown in detail in Algorithm 9.

As shown in Algorithm 10, the general structure of mapping and unmapping a memory region consisting of arbitrarily many kernel mappings is similar. The reason why we use this additional layer of code is that we want to keep track of the kernel mappings and the page table tree in user space as user space is responsible for setting up and modifying the page translation structures.

Also, the capability copies that are created during mapping are deleted when unmapping. This makes the fact that a mapping might need multiple kernel mappings transparent to the user of `libbarrelfish`.

Algorithm 9 Map arbitrary region in user space

```
function DO_MAP($vaddr$: virtual address at which to map,
                $frame$: frame capability to map,
                $offset$: offset in frame,
                $size$: size of mapping in bytes)
    $size \leftarrow$ ROUND_UP($size, BASE\_PAGE\_SIZE$)
    $vend \leftarrow vaddr + size$
    $pte\_count \leftarrow$ DIVIDE_ROUND_UP($size, BASE\_PAGE\_SIZE$)
    $entry \leftarrow$ GET_PT_ENTRY($vaddr$)
    ▷ Lookup the user space page table representation
    ▷ for the page table for $vaddr$
    $ptable \leftarrow$ GET_PTABLE($vaddr$)

    if mapping fits in leaf page table then
        create user space mapping representing kernel mapping
        KERNEL_MAP($ptable.cap, frame, entry, offset, pte\_count$)
        return $ERR\_OK$
    end if

    ▷ map pages in first leaf page table
    $rem\_pages \leftarrow pte\_count$
    $first\_count \leftarrow PTABLE\_SIZE - entry$
    create user space mapping representing kernel mapping
    KERNEL_MAP($ptable.cap, frame, entry, offset, first\_count$)
    $offset \leftarrow offset + first\_count * BASE\_PAGE\_SIZE$
    $vaddr \leftarrow vaddr + first\_count * BASE\_PAGE\_SIZE$

    ▷ map full leaf page tables,
    ▷ this could be replaced with mapping super pages, if supported
    repeat
        $fcopy \leftarrow$ CAP_COPY($frame$)
        $ptable \leftarrow$ FIND_PTABLE($vaddr$)
        create user space mapping representing kernel mapping
        KERNEL_MAP($ptable.cap, frame, entry, offset, PTABLE\_SIZE$)
        $rem\_pages \leftarrow rem\_pages - PTABLE\_SIZE$
        $offset \leftarrow offset + PTABLE\_SIZE * BASE\_PAGE\_SIZE$
        $vaddr \leftarrow vaddr + PTABLE\_SIZE * BASE\_PAGE\_SIZE$
    until remaining pages fit in one leaf page table

    ▷ map remaining pages
    create user space mapping representing kernel mapping
    $ptable \leftarrow$ FIND_PTABLE($vaddr$)
    KERNEL_MAP($ptable.cap, frame, entry, offset, rem\_pages$)
    return $ERR\_OK$
end function
```

Algorithm 10 Unmap arbitrary region in user space

```
function DO_UNMAP($vaddr$: virtual address at which to map,
                  $size$: size of mapping in bytes)
    $size \leftarrow$ ROUND_UP($size$, $BASE\_PAGE\_SIZE$)
    $vend \leftarrow vaddr + size$
    $pte\_count \leftarrow$ DIVIDE_ROUND_UP($size$, $BASE\_PAGE\_SIZE$)
    ▷ Lookup the user space page table representation
    ▷ for the page table and page for $vaddr$
    $ptable \leftarrow$ GET_PTABLE($vaddr$)
    $page \leftarrow$ GET_PAGE($ptable$, $vaddr$)

    if mapping fits in leaf page table then
        KERNEL_UNMAP($ptable.cap$, $page.cap$, $page.entry$, $pte\_count$)
        delete user space mapping representing kernel mapping
        return $ERR\_OK$
    end if

    ▷ unmap pages in first leaf page table
    $rem\_pages \leftarrow pte\_count$
    $first\_count \leftarrow PTABLE\_SIZE - entry$
    KERNEL_UNMAP($ptable.cap$, $page.cap$, $entry$, $first\_count$)
    delete user space mapping representing kernel mapping
    $offset \leftarrow offset + first\_count * BASE\_PAGE\_SIZE$
    $vaddr \leftarrow vaddr + first\_count * BASE\_PAGE\_SIZE$

    ▷ map full leaf page tables,
    ▷ this could be replaced with mapping super pages, if supported
    repeat
        $ptable \leftarrow$ GET_PTABLE($vaddr$)
        $page \leftarrow$ GET_PAGE($ptable$, $vaddr$)
        KERNEL_UNMAP($ptable.cap$, $page.cap$, $entry$, $PTABLE\_SIZE$)
        CAP_DESTROY($page.cap$)
        delete user space mapping representing kernel mapping
        $rem\_pages \leftarrow rem\_pages - PTABLE\_SIZE$
        $offset \leftarrow offset + PTABLE\_SIZE * BASE\_PAGE\_SIZE$
        $vaddr \leftarrow vaddr + PTABLE\_SIZE * BASE\_PAGE\_SIZE$
    until remaining pages fit in one leaf page table

    ▷ unmap remaining pages
    $ptable \leftarrow$ GET_PTABLE($vaddr$)
    $page \leftarrow$ GET_PAGE($ptable$, $vaddr$)
    KERNEL_UNMAP($ptable.cap$, $page.cap$, $entry$, $PTABLE\_SIZE$)
    CAP_DESTROY($page.cap$)
    delete user space mapping representing kernel mapping
    return $ERR\_OK$
end function
```

3.4 Memory Object Sharing

Utilizing the new KPI, we can implement memory region sharing that is robust in the presence of capabilities that are shared being revoked at any time. We show a proof-of-concept implementation of such memory region sharing in `libshm`, and show how this could be integrated into Barrelfish's memobjs.

3.4.1 Memory region sharing

We show `libshm` as a proof-of-concept implementation of memory region sharing in `libbarrelfish`. It has a simple API for creating, mapping, sharing, and deleting shared memory regions. To share memory between domains, this implementation requires an established communication channel between the participating domains. This channel is then used by `shm_init`, `shm_share`, and `shm_receive` to transfer the capabilities backing the shared memory region.

`libshm` decides whether to use a page table capability or a bunch of frame capabilities for sharing by a simple heuristic. If the region consists of a single frame capability, that frame capability is shared. If the region consists of multiple frame capabilities, they are all mapped in a page table that is not in the page table tree yet, and that page table capability is shared between domains and mapped in all participants' page table trees.

An improved implementation of these mechanisms could be encapsulated in the memobjs themselves. This would make the sharing mechanisms even more tightly integrated into the standard memory management interface.

3.4.2 Integration in libbarrelfish's memobjs

Using the ideas shown in Section 3.4.1, we can implement easy-to-use memory object sharing in Barrelfish by integrating the setup steps shown in `libshm` into the memobjs that are used to represent memory regions in Barrelfish's virtual address space management code.

Such a system can also decide (using a suitable heuristic) if it wants to share a page table or frames between domains, taking into account the cost of transferring the capabilities in the first place and the cost of updating the shared region among other things.

Given a capability system with reliable cross-core invocations we can even allow all domains participating in the sharing to update page tables that are shared in such a way. This gives us a memory management system that has properties similar to Corey's address ranges [2] but with a safe way of destroying shared regions.

What needs to be considered in such an implementation is how to do remote a TLB shootdown on all cores accessing the shared region if a mapping in a shared memory object is changed. On Barrelfish, such a shootdown would ideally involve the monitors on the respective domains' cores. That way, it is most likely possible to avoid a broadcast to all cores in the system when wanting to invalidate a TLB entry on a remote core.

4 Performance

We benchmarked the new kernel virtual memory interface on “ziger1” and “ziger2”, identical machines with 24 cores (4 CPUs with 6 cores each) that belong to the ETH Zürich Systems Group. The detailed hardware specifications can be found in Table 4.1.

CPU	4 x Six-Core AMD Opteron™ 8431
Frequency	2.4 GHz
L1 cache (per core)	64+64 KiB (i-cache and d-cache)
L2 cache (per core)	512 KiB
L3 cache (per CPU)	6 MiB (shared)
TLB (per core)	1024 4 KiB pages
Main memory (total)	16 GiB

Table 4.1: Hardware used for benchmarking

4.1 Methodology

In order to obtain similar measurements for the old and new kernel interface, we have written a small benchmarking utility running on a single core that repeatedly calls `vspace_map_one_frame` and `vregion_destroy` in `libbarrelfish` to map and unmap regions with sizes from one to 1024 pages.

To measure the time spent in the mapping and unmapping code, we have added cycle-counting code in the `libbarrelfish` map and unmap functions (cf. Algorithm 9 and Algorithm 10). To measure the overall cost, we’ve not instrumented the kernel calls themselves but rather just measure the cycles spent from start to finish of the whole function.

4.2 Expectations

We expected to see a performance increase for larger mappings due to the fact that the new interface needs only one syscall for mappings that do not span multiple leaf page tables and longer run-times for small unmap requests due to the additional work that has to be done to clear the kernel mapping belonging to the region that is being unmapped.

4.3 Results

4.3.1 Kernel interface micro-benchmarks

A first measurement we made compared the performance of only the kernel calls for both the old and new kernel interface. To do this, we added cycle-counting code around the kernel calls themselves in the pmap map and unmap functions rather than looking at the total amount of cycles spent in the pmap functions.

We expect a significant performance increase in the raw invocations for mapping more than one page, as we only have one system call instead of n system calls for mapping n pages, $n < 512$, assuming that we align the virtual address to map at the beginning of a leaf page table.

On the other hand, we expect unmap performance to take a hit for small numbers of pages due to the additional work that has to be done in order to enable selective TLB flushing.

Operation	#pages	Used CPU cycles			
		old KPI		new KPI	
		avg.	sdev.	avg.	sdev.
No-op	n/a	188	5.74	186	12.45
Invoke	n/a	277	12.96	272	15.49
Map	1	1330	423.72	1191	270.35
Map	32	17978	935.56	1339	147.33
Map	128	70156	2827.59	2787	198.99
Unmap	1	699	51.09	9518	609.08
Unmap	32	18554	413.43	9467	432.04
Unmap	128	77683	3658.19	9663	2102.92

Table 4.2: Average number of cycles spent in the kernel for mapping and unmapping 1, 32, and 128 pages.

We can see in Table 4.2 that both predictions are true to a point. Mapping pages is significantly faster for 128 consecutive pages using the new KPI: 2787 ± 199 cycles vs. 70156 ± 2827 cycles for the old KPI. The distribution of the cycles required to map 1, 32, and 128 pages using the old and the new KPI can be seen in Figure 4.1. For reference, the time spent in a null syscall is ≈ 190 cycles, and the time spent in a no-op capability invocation is ≈ 275 cycles (see Table 4.2). We also see that unmapping one page costs 699 ± 51 cycles using the old interface and 9518 ± 609 cycles using the new interface. However unmap performance remains stable for 1, 32 and 128 pages, confirming that the capability lookup is the dominating cost factor. The distribution for unmap is shown in Figure 4.2.

While these numbers show the cycles we spend in the kernel itself (plus the syscall and invocation overhead), just measuring the time spent in the kernel is not representative when handling regions that overlap multiple kernel mappings. Therefore these microbenchmarks should not be directly compared with the graphs in the next sections.

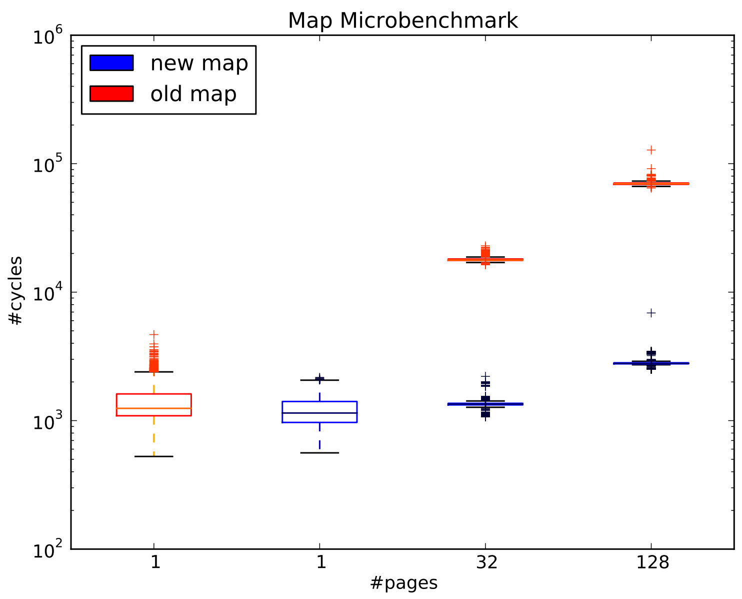

Figure 4.1: Cycle count distribution for mapping 1, 32, and 128 adjacent pages

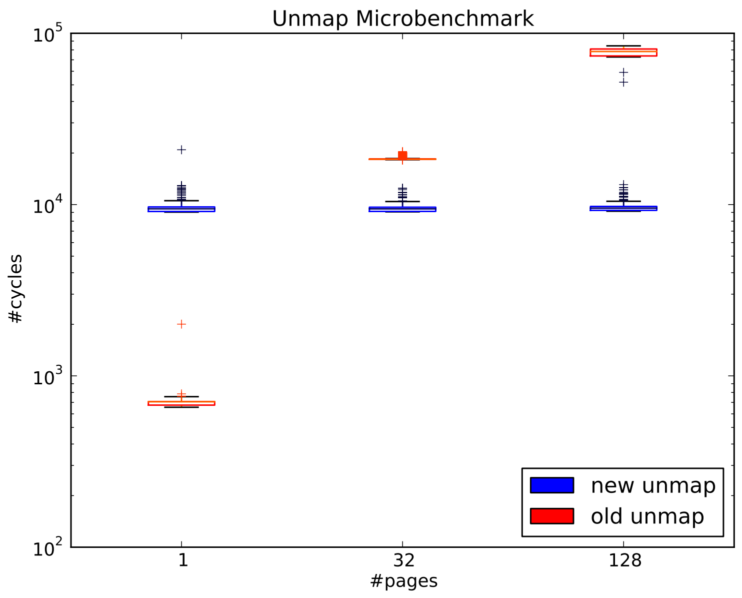

Figure 4.2: Cycle count distribution for unmapping 1, 32, and 128 adjacent pages

4.3.2 Mapping performance

As stated above, we expected mapping performance to increase when mapping more than one page in comparison to the old kernel interface. Figure 4.3 shows that while the performance per page for mapping a low number of consecutive pages is almost unchanged, the cost for mapping a single page in a larger consecutive mapping decreases significantly for larger mappings. Also the increased cost due to copying the capability for mapping a region with more than 512 pages is noticeable at first glance. The graph shows the performance of the old interface in red (the thin red lines are *average* ± *standard deviation*) and orange (the worst case in 1000 runs) and the performance of the new interface in blue (the thin lines are again *average* ± *standard deviation*) and cyan for the worst case in 1000 runs.

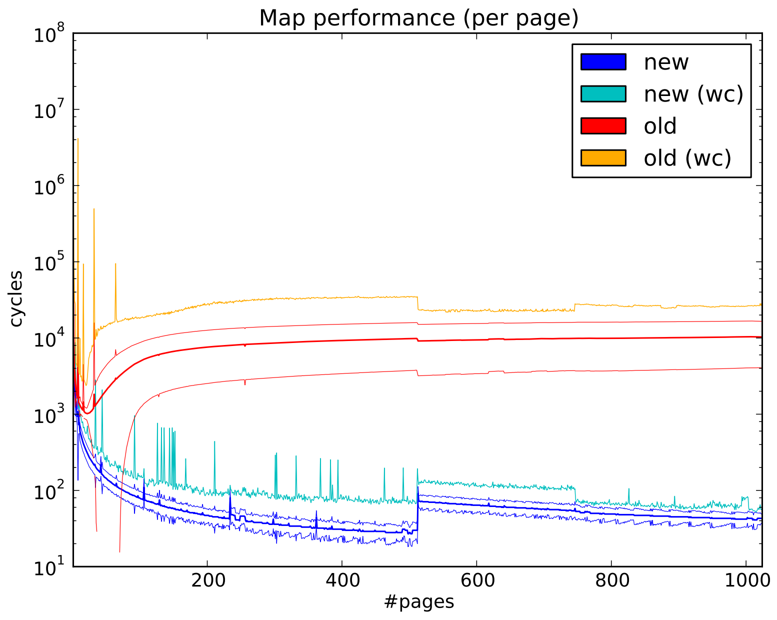

Figure 4.3: Average and worst case cycle counts per page for mapping 1 – 1024 adjacent pages.

Additionally, Figure 4.4 shows the total cycles for mapping 1–1024 pages. Again we can see that as soon as we map more than one page the performance increase is quite significant as we always only have one invocation, and we can also clearly see the point at which we have to create a copy of the capability as our mapping is now split into two kernel mappings (for 513 or more pages).

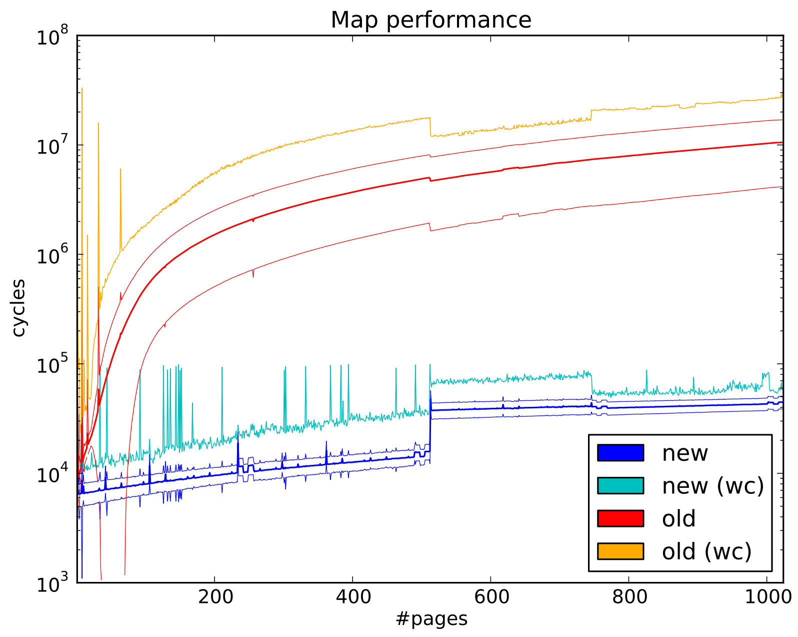

Figure 4.4: Average and worst case cycle counts for mapping 1 – 1024 adjacent pages.

4.3.3 Unmapping performance

In Figure 4.5 we see the performance of unmap per page for 1 to 1024 pages. Again, the red and orange are the values using the old interface and blue and light blue show the performance of unmap using the new interface.

With the new interface, the capability lookup by address dominates the cost, but having only one invocation for up to 512 pages offsets the cost of the lookup quite nicely, leading to overall better performance for mappings larger than ≈ 20 pages. Also, the cost of deleting the capability copy that was created when unmapping a region that is larger than 512 pages is clearly visible in the response time of unmap.

We also see in Figure 4.6 that the total cost of an unmap stays the same for 1 to 512 pages. The unmap performance graphs also reinforce the microbenchmark results: unmapping a small number of pages is much more expensive using the new KPI. This can be explained by the fact that unmap has to reconstruct the virtual address for selectively flushing the TLB, and the average cost for this address reconstruction $\approx 10500 \pm 3800$ cycles. Figure 4.7 shows a detailed breakdown of the cycle counts for virtual address reconstruction.

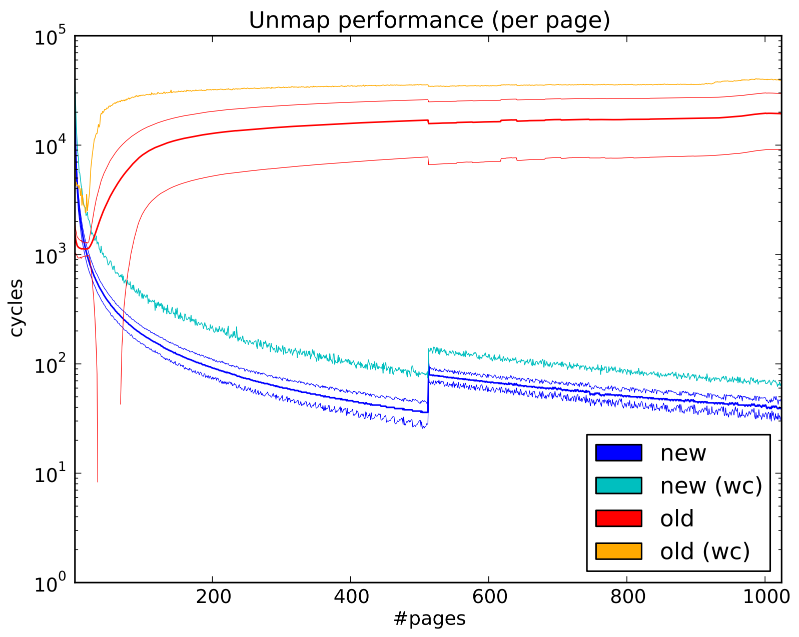

Figure 4.5: Average and worst case cycle counts per page for unmapping 1 – 1024 adjacent pages.

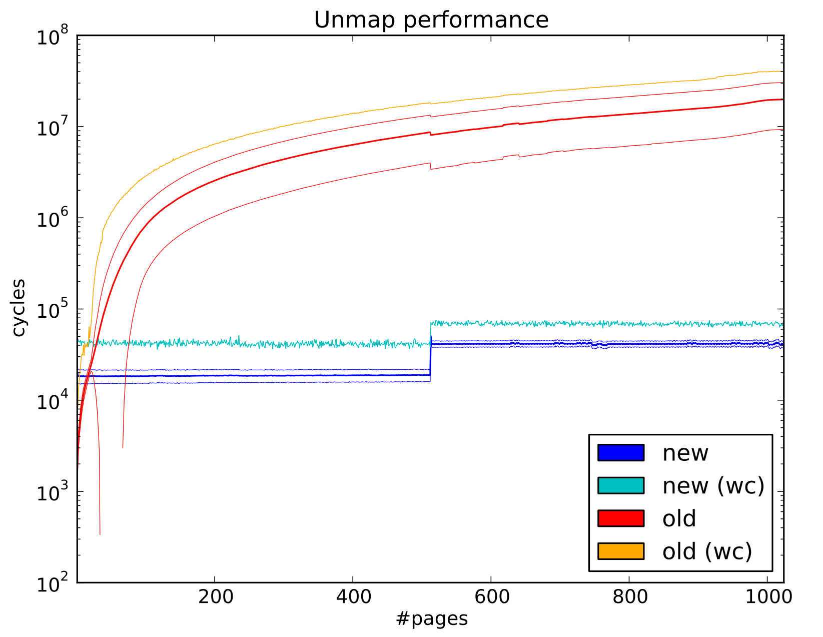

Figure 4.6: Average and worst case cycle counts for unmapping 1 – 1024 adjacent pages.

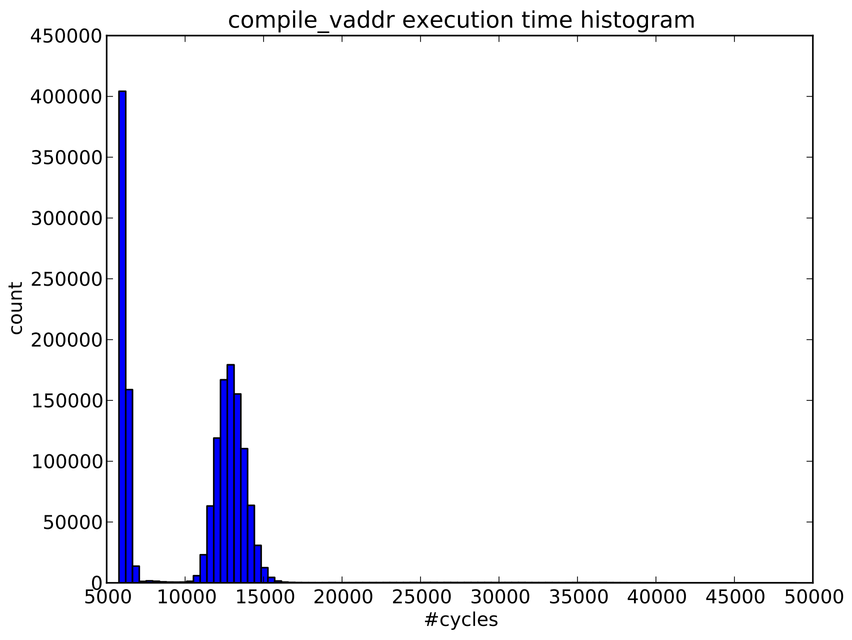

Figure 4.7: Histogram of response times of virtual address reconstruction.

4.3.4 Modified unmap implementation

While trying to explain the results in Figure 4.5 and Figure 4.6, we saw that doing a range query for the physical address mapped at the address to unmap and looking at all copies of the range query in order to find a matching kernel mapping contributes to the long runtime of unmap. We then changed the interface of unmap to take the capability representing the kernel mapping as an additional argument (see Algorithm 11).

Algorithm 11 Unmap a region (version with no kernel mapping lookup)

```
function UNMAP(ptable: the page table in which to unmap,
               entry: the entry in the page table,
               cap: the capability representing the kernel mapping to unmap,
               num_pages: the number of pages to unmap)
    $as, vaddr \leftarrow$ COMPILE_VADDR($ptable, entry$)
    $paddr \leftarrow$ GET_PADDR($ptable, entry$)
    $pte \leftarrow$ GET_ADDRESS($ptable$) $+ entry$
    if $cap.mapped\_pages \neq num\_pages$ then
        return $ERR\_MAP\_SIZE$
    end if
    $c \leftarrow 0$
    repeat
        ZERO($ptable$, $entry + c$)
        $c \leftarrow c + 1$
    until $c \geq num\_pages$
    FLUSH_TLB($as$, $vaddr$, $vaddr + num\_pages * BASE\_PAGE\_SIZE$)
    $cap.pte \leftarrow 0$
    $cap.mapped\_offset \leftarrow 0$
    $cap.mapped\_pages \leftarrow 0$
end function
```

While the code for map has not changed, we still see a performance increase that can probably be attributed to a less pressured cache, as unmap does not have to look up the capability representing the kernel mapping anymore (cf. Figure 4.8).

In Figure 4.9, we see that unmap is faster by a factor of about 1.5 when supplying the capability representing the kernel mapping as an additional argument.

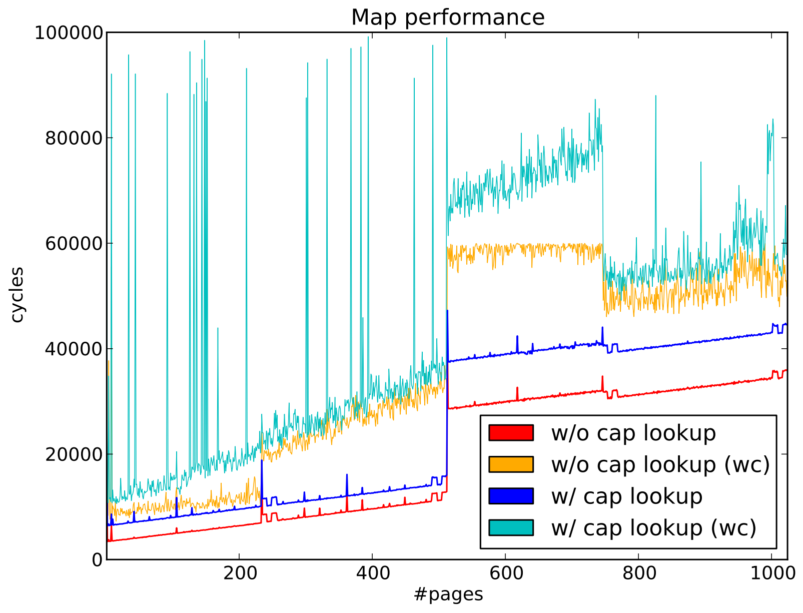

Figure 4.8: Comparison of the new kernel unmap interfaces: Average and worst case cycle counts for mapping 1 – 1024 adjacent pages.

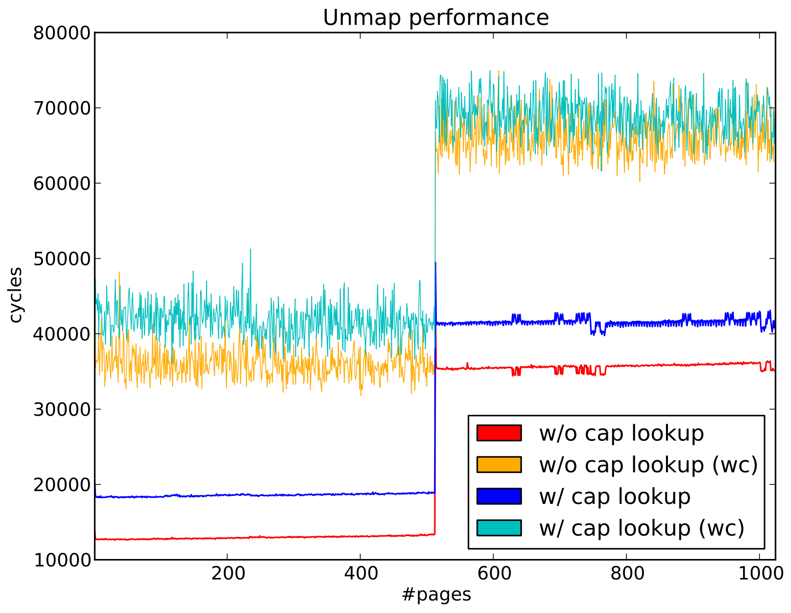

Figure 4.9: Comparison of the new kernel unmap interfaces: Average and worst case cycle counts for unmapping 1 – 1024 adjacent pages.

4.3.5 TLB flushing strategies

As mentioned earlier, when flushing the translation lookaside buffer we can choose different strategies. The two basic strategies are to either always flush the whole TLB or to selectively flush those TLB entries that correspond to a virtual address for which the mapping was changed. Using the benchmark described in Section 4.1, always selectively flushing the TLB over always flushing the whole TLB results in worse performance for both map and unmap (cf. Figure 4.10 and Figure 4.11).

	Used CPU cycles		
	avg.	sdev.	worst
reload cr3	297	284.51	1959
`invlpg`[1]	256	35.48	2730

Table 4.3: Micro-benchmark of TLB flushing strategies
[1] `invlpg` cycle counts are per page.

After seeing the microbenchmark results of the respective operations for selective and full TLB flushes (Table 4.3), we also looked at a hybrid TLB flushing approach where we use the selective flushing mechanism when unmapping a single page and flush the whole TLB when unmapping more than one page. We expected this hybrid scheme to give us overall better performance than always reloading the whole TLB.

However, while the hybrid strategy returns the unmapping response time to nearly the level of always flushing the whole TLB, map performance does not improve much. The improvement in map performance for regions consisting of more than 512 pages can be explained by the fact that we do not invalidate the TLB entries for the addresses where the capabilities are stored, and thus the capability copy is faster.

A better benchmark for TLB flushing strategies would involve standard workloads and would measure overall running times of these workloads in order to take the follow-up costs of TLB flushing – i.e. subsequent TLB misses – into account.

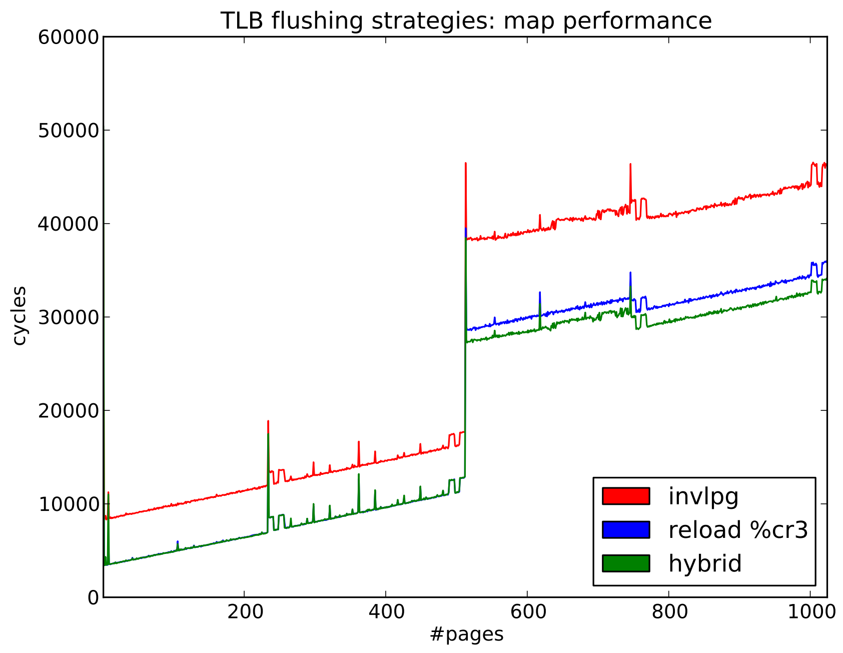

Figure 4.10: Mapping response times for different TLB flushing strategies.

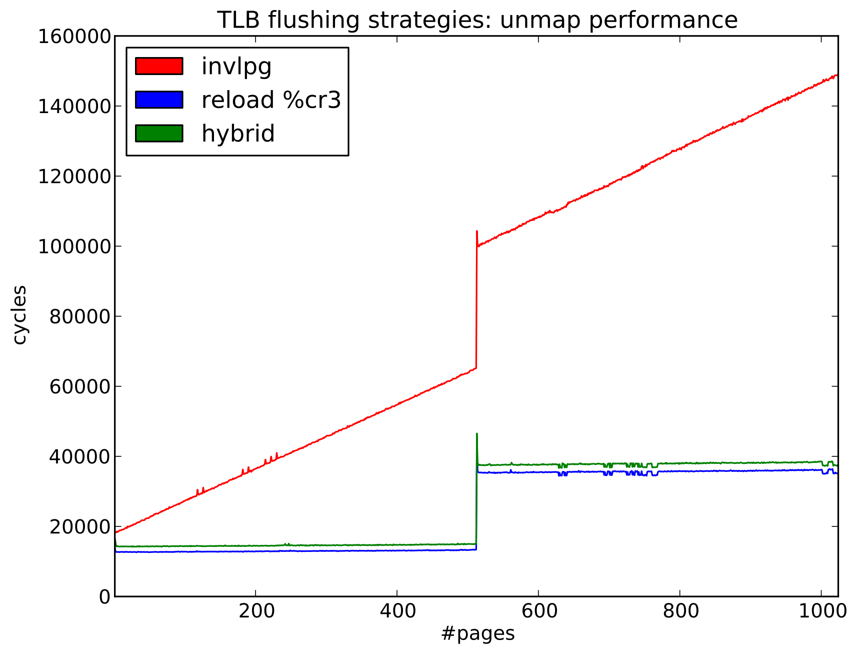

Figure 4.11: Unmapping response times for different TLB flushing strategies.

4.4 Analysis of our Methods

While the new kernel interface performs good overall, we see that the virtual address reconstruction and capability lookups are quite expensive (on the order of thousands of cycles). This behaviour gets worse, when the number of capabilities gets too large to fit the whole index tree into the CPU's cache. We try to mitigate one of the slow operations in unmap by supplying the mapping capability as additional function argument.

Another area where the results were not entirely expected are the different TLB flushing strategies. While doing an in-depth analysis of the results, we discovered that the average response times in the benchmark we show in Section 4.1 gets slower the more iterations we do. This can again be explained by the fact that the longer the benchmark runs, the more capabilities there are in the system. Figures 4.12 and 4.13 show how these slower times impact our results.

Additionally, Figure 4.14 shows the response times for unmapping one page and 100 pages over time. The marked increase in response time between iterations 50 and 70 is most likely around the time when the number of capabilities in the system ceases to fit in the cache.

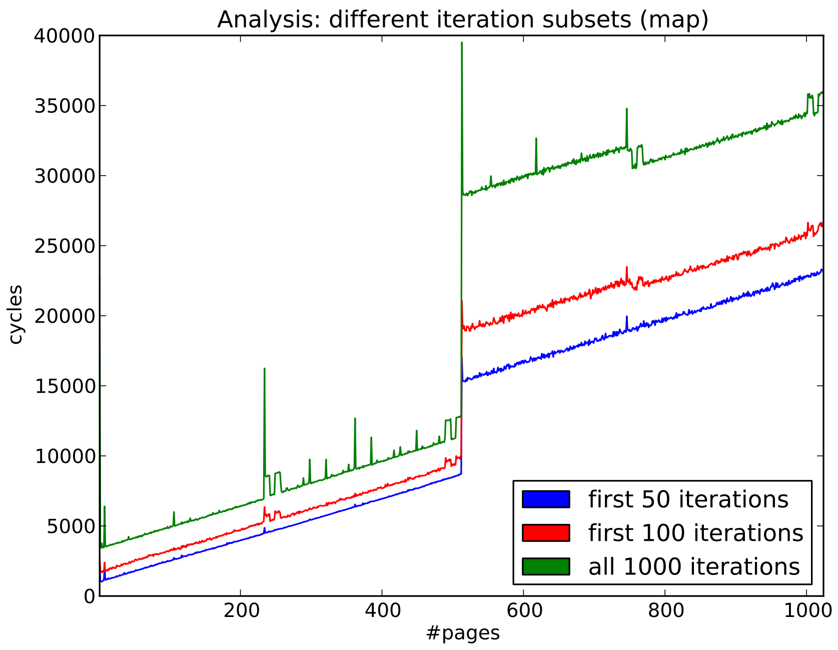

Figure 4.12: Mapping response times for the first 50, 100 and all 1000 iterations.

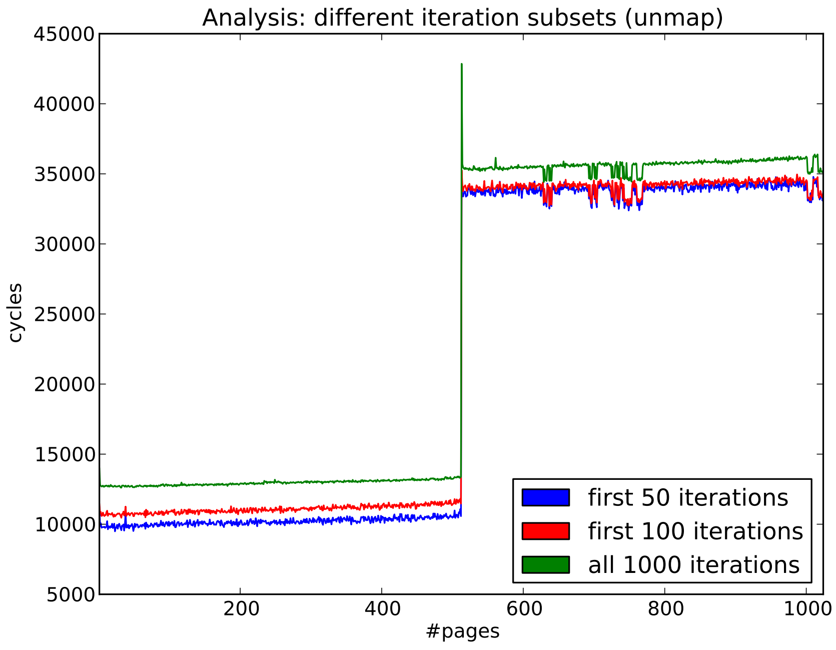

Figure 4.13: Unmapping response times for the first 50, 100 and all 1000 iterations.

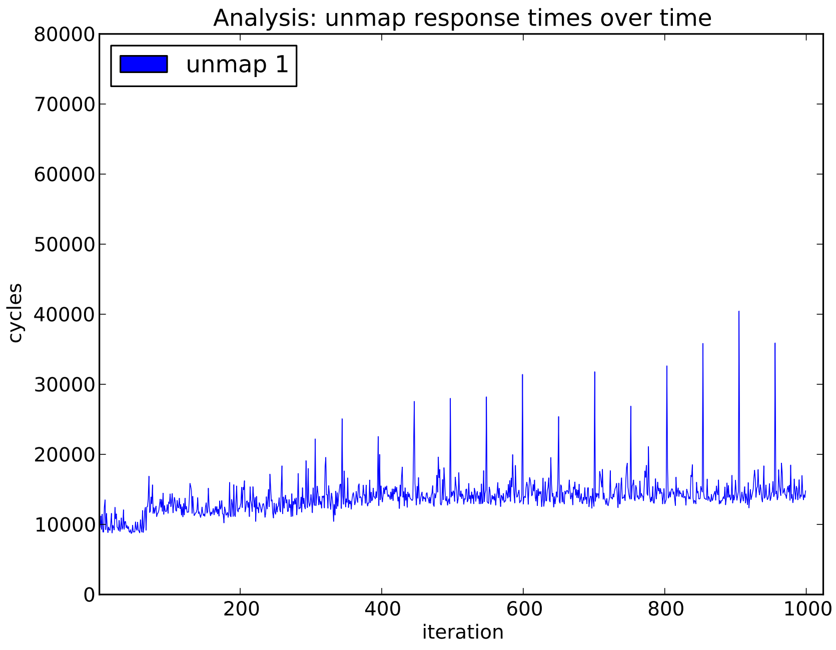

Figure 4.14: Unmapping response times for unmapping one page over time.

5 Conclusion

In this thesis I have described the need for a memory subsystem implementation that keeps more state in the kernel than what Barrelfish's current implementation does to implement safe, cross-core shared memory regions. I also show a design for such a kernel interface that relies on some functionality that the capability system has to provide – especially regarding cross-core capability operations – in order for shared memory to remain safe in a cross-core situation.

Additionally I show that an implementation of the presented kernel interface matches the old Barrelfish kernel interface in performance in most cases. However, as the capability system does not yet provide all the necessary functionality in order to guarantee safety in a cross-core situation, the new implementation still does not provide safe cross-core shared memory and I have not attempted to test the performance of the new system in a cross-core situation.

As Barrelfish now has a reliable way to unmap memory regions, and there were several parts of the system (most notably the acpica glue layer) that relied on the fact that unmapping was essentially a no-op, I had to change those parts to work with an unmap functionality that actually does remove mappings in the hardware page tables.

5.1 Future Work

While I have shown a new kernel interface that improves low level memory system performance, there remain a number of interesting problems on higher level sharing. A promising idea is to use Barrelfish's interface specification language and compiler and extend it in such a way that interface definitions get support for automatically translated pointers to shared memory regions. This would include adding functionality to the compiler so that it generates address translation and mapping tracking code to the generated interface glue code.

Seeing that memory is relatively cheap today, it might be interesting to compare the performance and memory requirements of the kernel interface I have presented with the performance and memory requirements of a traditional shadow page table implementation.

Additionally there is no particular reason why all capabilities that identify physical memory have to have a power of two size. It is possible to restrict physical memory capabilities sizes to a multiple of the base page size (usually 4 KiB) instead. This would make various operations easier If desired, another restriction that would be relatively easy to accommodate would be to restrict these capabilities to always be naturally aligned. Another related area that needs to be understood better is how memory can be reclaimed once the last copy of a capability identifying that memory is deleted. This would be considerably easier if we allow capabilities to have non-power-of-two sizes as we then could utilize range queries to figure out if a memory range still has a capability covering it.

A X86 Hardware Support for Memory Management and Address Translation

As described in Intel's Software Developer's Manual [8, ch.3], Intel's x86 (IA-32 and EM64T) architectures, hardware support for memory management, address translation, and memory protection is present in two forms: segmentation and paging. Segmentation provides isolation of code, data, and stack modules and is not optional. Paging provides a traditional demand-paged virtual memory system which can be used for isolation as well. However, unlike segmentation, paging can be disabled completely. Most operating systems choose not to do so, as it is hard to work with limited amounts of physical memory and no demand paging.

A.1 Segmentation

Memory segmentation works by dividing the processor's addressable memory space (the linear address space) into smaller protected address spaces, the segments. Thus memory addresses in the processor are logical addresses (also called far pointers) that consist of a segment selector and an offset. The segment selector is a unique identifier for the segment which contains an offset into the global descriptor table (GDT). Using that offset, the processor retrieves a segment descriptor that contains the base and size of the segment as well as the access rights and privilege level for that segment. The linear address is then computed by adding the offset of the logical address to the base of the segment.

If paging is disabled, linear addresses are directly mapped to the physical address space, i.e. the range of addresses that the processor can generate on its address bus.

There are several different usage models for segmentation. The most basic model is called basic flat segmentation and hides most of the segmentation system from the operating system and applications. In this model, applications and operating system have access to a contiguous unsegmented address space.

The next level of usage is called protected flat segmentation and differs from basic flat segmentation by having segment limits that restrict program access to the address range that can actually contain physical memory.

The usage model that makes full use of the capabilities of the segmentation hardware is called multi-segment model. In this model each application has its own set of segments which – if so desired – can be shared among several cooperating applications.

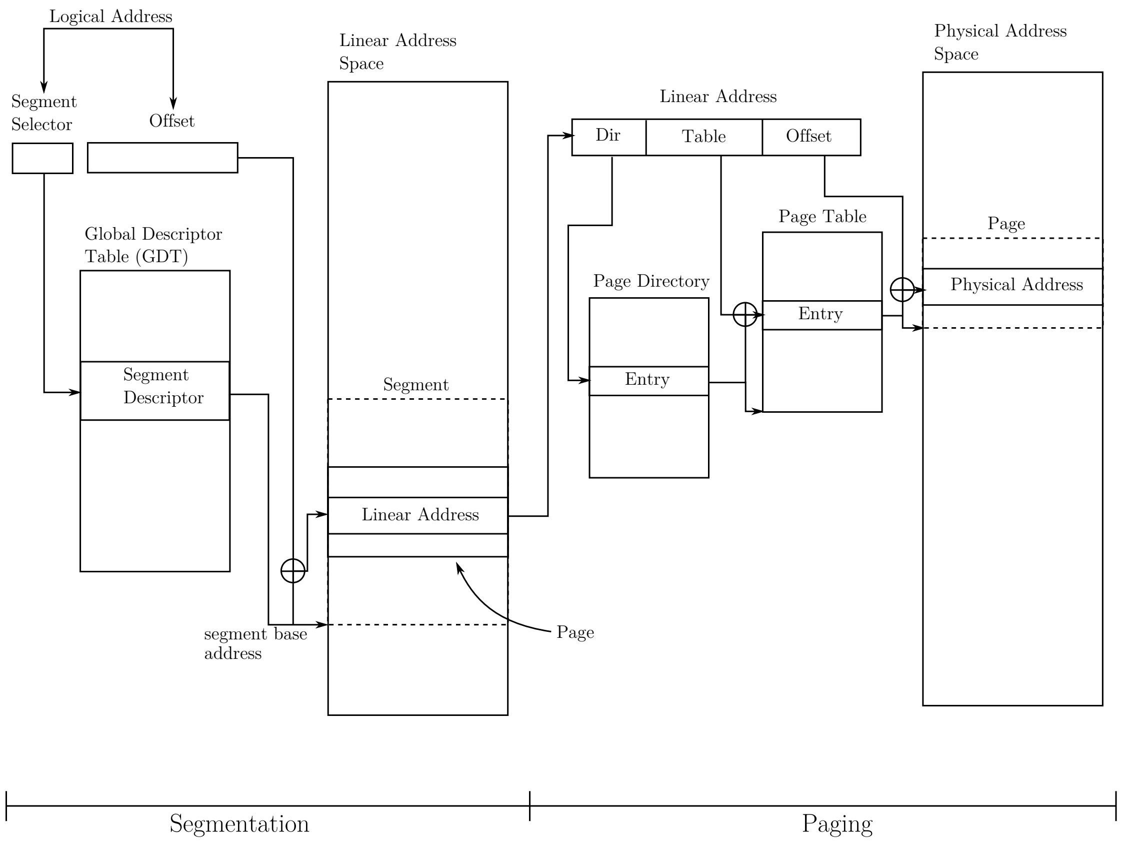

Figure A.1: Linear address lookup (from [8, Vol.3A,p.3-2])

A.2 Paging

As multitasking systems usually define a linear address space that cannot be mapped directly to physical memory due to its size, demand paging ("paging") virtualizes the linear address space, thus producing the more familiar "virtual addresses". The virtualization of the linear address space is handled by the processor's paging hardware. Using paging we can simulate a large linear address space with a small amount of physical memory and some disk storage. Using paging, each segment is split into pages of 4 KiB in size that are either stored in physical memory or on disk. The operating system has to maintain a page directory and a set of page tables to keep track of all the pages in the system. When a program attempts to access a linear address, the processor uses the page directory and page tables to translate the (virtual) linear address into a physical address and uses the generated physical address to perform the actual memory access (cf. Figure A.1).

If a page corresponding to a memory access (using a virtual address) is not in physical memory, the processor generates a page-fault exception, thus interrupting the program trying to access memory. The operating system then reads the missing page from disk (or allocates a new region of physical memory), installs that page in the appropriate page tables and resumes execution of the program.

As the paging mechanism described above is similar for 32 bit and 64 bit x86 processors, we have so far ignored the various subtle differences. In fact, there are three distinct paging models for x86 processors: standard 32-bit paging, PAE paging, and IA-32e paging.

Standard 32-bit paging uses 32-bit linear addresses and has a page directory with 1024 entries that point to page tables containing 1024 page entries. Standard 32-bit paging can support physical page extension (PSE) that allows the physical addresses to be up to 40 bits wide. Standard 32-bit paging allows $4\,KiB$ and $4\,MiB$ pages.

PAE (physical address extension) paging is an extension of 32-bit paging that allows physical addresses to be 52 bits wide. When PAE is enabled, the processor maintains a set of four PDPTE registers that are loaded from a 32 byte page directory pointer table. These PDPTE registers are then used to translate linear addresses. Using PAE, all page table entries are 64 bits wide, and the system supports page sizes of $4\,KiB$ and $2\,MiB$.

IA-32e paging is used on 64-bit processors and translates 48-bit linear addresses to 52-bit physical addresses. IA32-e uses four levels of page tables with 64-bit entries to translate addresses. IA32-e mode can support $4\,KiB$, $2\,MiB$, and $1\,GiB$ pages. Of those page sizes, support for $4\,KiB$ and $2\,MiB$ pages is mandatory.

Table A.1 gives an overview of the paging structures and their usage with the three paging modes.

Table A.1: Intel Paging Structures, from [8, Vol. 3A, p.4-9]

Paging Structure	Paging Mode	Physical Address of Structure	Relevant Virtual Address Bits	Page Mapping[1]
PML4 Table	32-bit, PAE[2]	N/A		
	IA-32e[3]	CR3[4]	47:39	N/A
Page-directory Pointer Table (PDPT)	32-bit	N/A		
	PAE[2]	CR3[4]	31:30	N/A
	IA-32e[3]	PML4 entry	38:30	1 GiB page[5]
Page directory	32-bit	CR3[4]	31:22	4 MiB page[6]
	PAE[2], IA-32e[3]	PDPT entry	29:21	2 MiB page
Page table	32-bit	PD entry	21:12	4 KiB page
	PAE[2], IA-32e[3]		20:12	4 KiB page

[1]In this column is noted what size pages (if any) can be mapped in that type of page table entry. [2]PAE stands for Physical Address Extension, which extends physical addresses on a 32 bit processor from 32 to 52 bits. [3]IA-32e is Intel-speak for the 64bit paging mode. [4]CR3 is a special purpose register on x86 that contains the address to the current application's root page table. [5]Support for 1 GiB pages is processor specific (1 GiB page support is indicated in the return value of the `cpuid` instruction). [6]Support for 4 MiB pages on 32bit is processor specific and must be explicitly enabled in CR4.

A.3 Privilege Levels

The segment protection mechanism uses four privilege levels numbered from 0 to 3. Figure A.2 shows that these levels can be interpreted as rings of protection. The center (the "most" protected ring) is used for critical software, usually the kernel of an operating system. The two rings between the outermost and the central ring can be used for semi-critical software (e.g. operating system services). A system that does only use two protection levels should use levels 0 and 3 [8, sec.5.5].

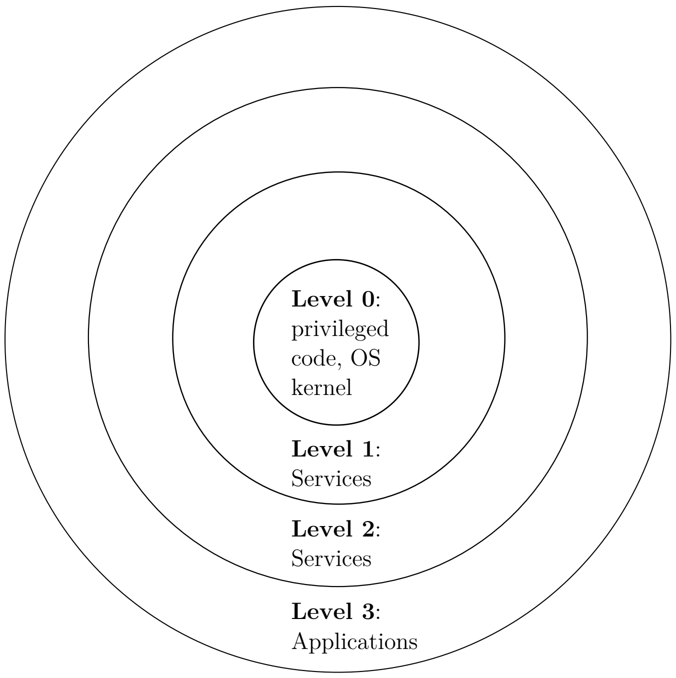

Figure A.2: The protection rings

Bibliography

[1] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 29–44, New York, NY, USA, 2009. ACM.

[2] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, and Zheng Zhang. Corey: an operating system for many cores. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 43–57, Berkeley, CA, USA, 2008. USENIX Association.

[3] David R. Cheriton and Kenneth J. Duda. A caching model of operating system kernel functionality. *SIGOPS Oper. Syst. Rev.*, 29(1):83–86, January 1995.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* MIT Press and McGraw-Hill, 2 edition, 2001.

[5] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: an operating system architecture for application-level resource management. *SIGOPS Oper. Syst. Rev.*, 29(5):251–266, December 1995.

[6] Steven M. Hand. Self-paging in the Nemesis operating system. In *Proceedings of the third symposium on Operating systems design and implementation*, OSDI '99, pages 73–86, Berkeley, CA, USA, 1999. USENIX Association.

[7] Norman Hardy. KeyKOS architecture. *SIGOPS Oper. Syst. Rev.*, 19(4):8–25, October 1985.

[8] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3: System Programming Guide.* Number 325384-042US. March 2012.

[9] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[10] Ihor Kuz. *L4 User Manual — API Version X.2*. Embedded, Real-Time and Operating Systems Program, NICTA, Jun 2004. Available from `http://www.disy.cse.unsw.edu.au/Software/L4`.

[11] Dominik Menzi. Support for heterogeneous cores for Barrelfish. Master's thesis, ETH Zürich, July 2011.

[12] Mark Nevill. An evaluation of capabilities for a multikernel. Master's thesis, ETH Zürich, May 2012.

[13] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. Rethinking the library os from the top down. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '11, pages 291–304, New York, NY, USA, 2011. ACM.

[14] Richard Rashid, Avadis Tevanian, Michael Young, David Golub, Robert Baron, David Black, William Bolosky, and Jonathan Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. *SIGOPS Oper. Syst. Rev.*, 21(4):31–39, October 1987.

[15] Mark E. Russinovich, David A. Solomon, and Alex Ionescu. *Windows® Internals*. Microsoft Press, 5 edition, June 2009.