LEHRSTUHL FÜR BETRIEBSSYSTEME

Eine generische Speicherverwaltung mit Hilfe von Seitentabellen für ein minimalistisches Betriebssystem

A generic memory management with paging for a minimalistic operating system

Steffen Vogel
post@steffenvogel.de
Matrikelnummer: 304957

Bachelorarbeit
an der
Rheinisch-Westfälischen Technischen Hochschule Aachen
Fakultät für Elektrotechnik und Informationstechnik
Lehrstuhl für Betriebssysteme

RWTH AACHEN UNIVERSITY

Betreuer: Dr. rer. nat. Stefan Lankes

Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten und nicht veröffentlichten Schriften entnommen sind, sind als solche kenntlich gemacht. Die Arbeit ist in gleicher oder ähnlicher Form noch nicht als Prüfungsarbeit eingereicht worden.

Aachen, den 20. Juni 2014

Kurzfassung

Diese Arbeit stellt einen neuen Ansatz zur Implementierung von Paging mit Hilfe von selbstreferenzierenden Seitentabellen vor. Dabei wurde Wert auf eine einfache und zugleich effiziente Umsetzung gelegt. Als Basis wurde das minimalistische Forschungsbetriebssystem „MetalSVM“ genutzt. Die Selbstreferenzierung ermöglicht es Seitentabellen auf eine einfache Art im virtuellen Adressraum einzublenden ohne dafür weiteren Speicherplatz zu belegen. Die vorgestellte Implementierung ist sowohl mit der 32- als auch 64 Bit-Version der x86-Architektur von Intel kompatibel. Dies ermöglichte es, die zuvor getrennten Implementierungen zusammenzufassen. Der daraus entstandene Code ist kürzer, leichter verständlich und wartbarer. Im Zuge dessen wurden auch weitere Teile der Speicherverwaltung überarbeitet. Es werden einfache Methoden zur Organisation des virtuellen Adressraums (VMAs) sowie eine Methode zur dynamischen Verwaltung des Heap Speichers vorgestellt. Die neue Speicherverwaltung wurde sowohl auf emulierter als auch echter Hardware getestet. Für den Test auf echter Hardware wurde ein Ablauf entwickelt, der die Zeit zwischen den Testzyklen minimiert. Abschließend wurde ein Benchmark eingesetzt um Charakteristika des Speichersystems zu bestimmen.

Stichwörter: LfBS, Bachelorarbeit, Paging, x86, MetalSVM, Speicherverwaltung, Selbstreferenzierung, minimalistisch

Abstract

This thesis presents a new approach for implementing paging with the aid of self-mapped page tables. For this a simple as well as efficient implementation has been emphasized. As base a minimal research operating system named "MetalSVM" was used. The self-referred nature of this approach maps the page tables inside the virtual address space without requiring additional memory. The presented solution is both compatible with the 32- and 64 bit version of Intel's x86-architecture. This allowed merging two former separated implementations into a new universal one. Thus lead to shorter, easier comprehensible and maintainable code base. At the same time other parts of the memory management subsystem have been revised and extended. A method for tracking free portions of the virtual address space was introduced (VMAs). To eliminate the waste of unusable memory a simple allocation scheme is used. This buddy system allows dynamic memory allocation inside the kernel. The revised memory management subsystem was both tested on real and emulated systems. For this a toolchain was assembled to ramp up the turnaround time between test cycles. Finally a memory benchmark was used to determine characteristics of the memory architecture.

Keywords: LfBS, Bachelor thesis, Paging, x86, MetalSVM, Memory Management, Self-mapped, Recursive, minimalistic, simple

Inhaltsverzeichnis

Abbildungsverzeichnis viii

Tabellenverzeichnis ix

1. Einführung 1

2. Paging 4

2.1. Grundlagen . . . 4
2.1.1. Eigenschaften . . . 5
2.1.2. Fragmentierung . . . 6
2.1.3. Mathematische Betrachtungsweise . . . 6
2.2. Die x86-Architektur . . . 9
2.2.1. Historische Entwicklung . . . 9
2.2.2. Seitentabellen . . . 10
2.2.3. Flags . . . 12
2.2.4. Multilevel-Paging . . . 13
2.2.5. Kanonische Adressen . . . 15
2.2.6. Translation Lookaside Buffer . . . 16
2.3. Selbstreferenzialität . . . 19
2.3.1. Beispiel . . . 20
2.3.2. Vorteile . . . 23
2.3.3. Nachteile . . . 24
2.4. Implementierung . . . 26
2.4.1. Operationen auf Seitentabellen . . . 26
2.4.2. Seitentabellen als Baum . . . 27
2.4.3. Durchläufe durch Seitentabellen . . . 27
2.4.4. MetalSVM . . . 29
2.4.5. Beispiel eines einfachen Durchlaufes . . . 31
2.4.6. Kopieren und Löschen von Seitentabellen . . . 32
2.4.7. Beispiel eines Speicherabbildes . . . 32
2.5. Vergleich weiterer Architekturen . . . 34
2.5.1. Inverse Seitentabelle . . . 34
2.5.2. Software TLB . . . 34

3. Virtual Memory Areas **37**
3.1. Grundlagen . . . 37
3.1.1. Bezug zum PA / Paging . . . 37
3.2. Implementierung . . . 38
3.2.1. Henne-Ei-Problem . . . 39
3.2.2. MetalSVM . . . 39

4. Heap Speicher **42**
4.1. Grundlagen . . . 42
4.1.1. Buddysystem . . . 43
4.2. Implementierung . . . 44

5. Toolchain und Hardware **48**
5.1. Quick Emulator . . . 48
5.1.1. Monitoring Konsole . . . 48
5.1.2. Universal Asynchronous Receiver Transmitter . . . 49
5.1.3. 32/64-Bit-Debugging . . . 49
5.2. Testsystem . . . 50
5.2.1. Hardware . . . 50
5.2.2. Testablauf . . . 51
5.2.3. Performance Monitoring Counter . . . 52

6. Benchmark und Tests **55**
6.1. Tests . . . 55
6.2. Membench . . . 56

7. Zusammenfassung und Ausblick **63**

A. Quellcode **65**

B. Kernel-Log **69**

C. Werkzeuge **74**
C.1. GDB Longmode Patch . . . 74
C.2. iPXE Konfiguration . . . 75
C.3. HWreset . . . 76

D. Alter Ansatz **78**

E. Speicherabbild **79**

Literaturverzeichnis **80**

Abkürzungsverzeichnis **83**

Abbildungsverzeichnis

2.1. Einfache Ansicht des Paging Konzeptes. 4
2.2. Mathematische Ansicht der Memory Management Unit. 7
2.3. Der Weg einer Adresse in der x86-Architektur. 9
2.4. Ablauf eines Pagetable-Walks. 11
2.5. Virtuelle Adresse und Tabelleneintrag im Protected-Mode. 12
2.6. Virtuelle Adresse und Tabelleneintrag im Longmode. 12
2.7. Vierstufiges Multilevel-Paging im Longmode. 14
2.8. 4 MiB Hugepage im Protected-Mode. 15
2.9. Das Loch des virtuellen Adressraums. 17
2.11. Longmode-Paging mit selbstreferenzierender PML4-Tabelle. 20
2.12. Adressen der Seitentabelleneinträge durch Selbstreferenzierung. . . . 23
2.13. Ein orange hervorgehobener Teilbaum. 27
2.14. Baumdurchläufe. 28
2.15. Schema einer inversen Seitentabelle mit Hashfunktion H. 36
2.16. Adressübersetzung mit Software-verwaltetem TLB. 36

3.1. VMA-Liste eines Prozesses im User-Space. 39

4.1. Vergleich des binären Buddysystems mit den DIN Papierformaten. . 44
4.2. Die verkettete Liste des Buddysystems. 45

6.1. Zugriffsmuster des Membench. 56
6.2. Latenz des Membench. 58
6.3. Cache-Misses des Membench. 59
6.4. Daten-TLB-Misses des Membench. 61
6.5. Dauer von `map_region()` in Abhängigkeit der Seitenanzahl. 62

C.1. Schaltplan und Layout von HWreset. 77

D.1. Alter Ansatz des Pagings von MetalSVM im Protected-Mode. . . . 78
D.2. Speicherabbild von MetalSVM im Longmode. 79

Tabellenverzeichnis

2.1. Vergleich des Pagings zwischen Protected- und Longmode. 10
2.2. Die Flags des Pagings im Longmode. 13
2.3. Größe der Seitentabellen für das Einblenden einer Seite. 14
2.4. TLB-Parameter der Haswell Mikroarchitektur von Intel [9], [11]. . . 18
2.5. Basisadressen der Seitentabellen im virtuellen Adressraum. 21
2.6. Page-Dump nach einer User-Space-Anwendung von MetalSVM. . . 33
2.7. Vergleich des Pagings gängiger Architekturen. 35

3.1. Die VMA-Regionen von MetalSVM. 40

5.1. Die Standard IO-Ports des UART-Transceivers. 50
5.2. Hardware des Testsystems. 51
5.3. TLB und Cachedetails des Testsystems. 51
5.4. Auszug der relevanten Performance-Monitoring-Ereignisse. 54
5.5. Performance Monitoring Counters des Testsystems. 54

6.1. Gemessene Latenzen der Speicherhierachie. 57

1. Einführung

Diese Arbeit widmet sich der Implementierung einer Speicherverwaltung für ein minimalistisches Betriebssystem (BS). Als Grundlage für diese Implementierung dient das am Lehrstuhl für Betriebssysteme (LfBS) entwickelte Forschungsbetriebssystem MetalSVM[1]. MetalSVM[2] ist ein leichtgewichtiges BS, das als Hypervisor für den Single-chip Cloud Computer (SCC) von Intel entwickelt wurde. Der SCC ist ein experimentelles Prozessordesign, welches 48 Pentium Kerne auf einem Chip vereint. Jeder dieser 48 Kerne führt dabei eine eigene Instanz des BS aus und ist über ein On-Chip-Netzwerk mit anderen Kernen und gemeinsamem Speicher vernetzt.

MetalSVM gehört zur Gruppe der „Lightweight Kernel Operating Systems“ (LWK). Durch seinen einfachen Aufbau eignet es sich daher gut für Benchmarks und Tests [18]. Seiteneffekte und Hintergrundrauschen lassen sich weitestgehend eliminieren. Weitere Vertreter der LWKs sind z.B. das in Assembler implementierte Baremetal OS[3] und Kitten[4].

Ursprünglich unterstützte MetalSVM ausschließlich die vom SCC verwendete 32-Bit-Version der Intel Architektur. Mit der zunehmenden Verbreitung der 64-Bit-Erweiterung wurde MetalSVM auch im Hinblick auf die Nutzung mit 64-Bit-Prozessoren wie bspw. Intels Many Integrated Core Architecture (MIC) angepasst und weiterentwickelt. Mit der 64-Bit-Erweiterung der Architektur konnte die bis dahin bestehende 4 GiB-Grenze des Arbeitsspeichers (ASP) überwunden werden. Dies ist für Anwendungen mit großen Speicheranforderungen wie z.B. Datenbanken oder Videobearbeitung von Vorteil. Neben der Breite des Adressbusses wurde gleichzeitig auch die Breite der Datenwörter verdoppelt. Davon profitiert unter anderem die Integer Arithmetik großer Datentypen wie sie bspw. bei der Verschlüsselung oder Multimedia Anwendungen zum Einsatz kommt.

Im Vordergrund der Arbeit steht die Unterstützung von Paging im 64-Bit-User-Space. Die Arbeit baut damit auf der schon existierenden x86-64-Portierung von MetalSVM auf. Die bisherige Implementierung nutzte jeweils separate Implementierung für die 32- bzw. 64-Bit-Version der Architektur. Auch die Initialisierung von MetalSVM wurde zweigleisig ausgeführt. Im Hinblick auf die Leichtgewich-

[1] Der Name setzt sich aus „Baremetal“ und „Shared Virtual Machine“ zusammen.
[2] `http://www.lfbs.rwth-aachen.de/content/765`.
[3] `http://www.returninfinity.com/baremetal.html`.
[4] `https://software.sandia.gov/trac/kitten`.

tigkeit des BS sollte es das Ziel sein, diese getrennten Teile zu kombinieren. Ein somit kürzerer Code ist leichter lesbar und verständlich. Er enthält in der Regel weniger Fehler und ist einfacher zu Warten. Im Zuge dessen wurden auch weitere Teilbereiche der Speicherverwaltung (engl. Memory Management) umstrukturiert und ergänzt:

Paging Paging übersetzt Adressen des virtuellen Adressraums (VA) in physikalische Adressen. Es abstrahiert damit den Speicherzugriff für den Programmierer. Es isoliert Prozesse und schützt die Daten des Kernels vor unberechtigten Zugriffen.

Für die 64-Bit-Portierung von MetalSVM musste das bisher zweistufige Paging auf vier Ebenen erweitert werden. Kapitel 2 stellt einen neuen Ansatz vor, der mit Hilfe von selbstreferenzierenden Seitentabellen einen einfachen Zugriff auf alle Paging-Strukturen ermöglicht.

Virtual Memory Areas Für das Verwalten des Speichers benötigt das BS die Kenntnis, welche Speicherbereiche bereits belegt und welche noch zur Verfügung stehen. Einfacherweise könnte man hierfür die Seitentabellen verwenden, da sie diese Informationen bereits enthalten. Jedoch ist das Durchsuchen dieser Tabellen, insbesondere nach freien Bereichen, sehr ineffizient.

Kapitel 3 beschreibt eine alternative Datenstruktur, die dieses Problem löst. Mit Hilfe von verketteten Listen werden zusammenhängende Speicherbereiche des virtuellen Adressraums verwaltet und mit Eigenschaften versehen. Das Konzept der VMAs wird mit der für den physikalischen Adressraum genutzten Bitmap verglichen.

Dynamische Verwaltung des Heap Mit Hilfe von Paging können nur ganze Speicherseiten eingeblendet werden. Werden nur Teile dieser Seiten benötigt, wird der restliche Speicherplatz verschwendet. Der Heap fasst diese kleinen Speicherbereiche zusammen und vermeidet so die Fragmentierung des Speichers.

Kapitel 4 beschreibt eine einfache dynamische Speicherverwaltung für den Heap.

Tests und Benchmarks Abschließend wurde die Implementierung mit Hilfe von Testroutinen und einem Benchmark überprüft. Dazu wurde die Implementierung auf echter Hardware getestet, die im Kapitel 5.2 vorgestellt wird. Im letzten Kapitel wird der Membench Benchmark benutzt, um einige Kenndaten der Speicherarchitektur des Testsystems zu bestimmen.

Kapitel 2, 5 und 6 beschreiben hardwarespezifische Funktionen und Eigenschaften. Daher beziehen sie sich zwangsläufig auf die in dieser Arbeit beschriebene

x86-Architektur von Intel. Die übrigen Kapitel 3 und 4 bauen auf dem zuvor beschriebenen Paging auf. Sie sind daher hardwareunabhängig und können auch auf andere Systeme übertragen werden.

2. Paging

2.1. Grundlagen

Paging (dt. Kachelverwaltung) ist ein wichtiger Bestandteil der virtuellen Speicherverwaltung. Es bezeichnet das Zerteilen des Speichers in viele, jeweils gleich große, Kacheln. Dies gilt sowohl für den existierenden Arbeitsspeicher (ASP) als auch für den virtuellen Adressraum (VA). Die Memory Management Unit (MMU) muss den *Seiten* aus dem virtuellen Adressraum (VA) je eine *Kachel* des physikalischen Adressraums (PA) zuordnen (siehe Abbildung 2.1). Im folgenden Sprachgebrauch wird Kachel (engl. page frame) einen Teil des PA und Seite (engl. page) einen Teil des VA bezeichnen [24]. Jede Kachel wird durch eine Page Frame Number (PFN) identifiziert. Dieser Index beschreibt die Position der Kachel innerhalb des ASP. Seiten besitzen mit der Virtual Page Number (VPN) einen ähnlichen Index, der ihrer Position im VA entspricht.

Eine Seite stellt die kleinste Einheit dar, die durch Paging verwaltet werden kann. Für gängige Architekturen sind die Kacheln in der Regel nur wenige Kilobyte groß.

Paging erfordert in der Regel das Vorhandensein einer MMU, die die Adressumsetzung übernimmt. Häufig ist diese eine eigenständige Hardwareeinheit, die direkt

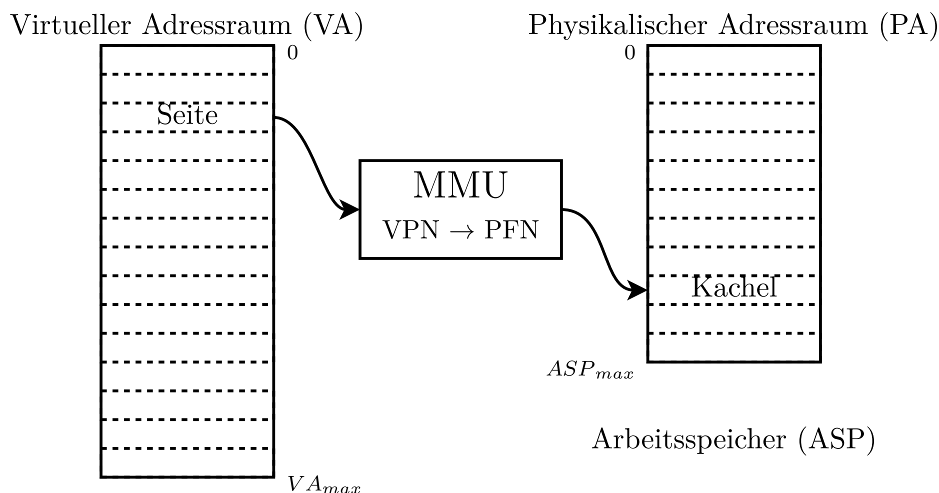


Abbildung 2.1.: Einfache Ansicht des Paging Konzeptes.

in die CPU integriert ist[1]. In eingebetteten Systemen und kleineren Mikroprozessoren wird häufig kein Paging eingesetzt. Gründe dafür sind geringere Kosten, eine geringere Leistungsaufnahme oder strenge Echtzeit-Anforderungen, die sich durch das Einsparen einer MMU erzielen lassen[2].

2.1.1. Eigenschaften

Paging besitzt eine Reihe vorteilhafter Eigenschaften, die dazu geführt haben, dass es sich in heutigen Rechnerarchitekturen weitestgehend durchgesetzt hat:

Isolierung von Prozessen Um die Sicherheit heutiger Anwendungen zu garantieren, ist es nötig, Prozesse untereinander zu isolieren. So darf z.B. das Mailprogramm keinen Zugriff auf die Daten des Online-Banking erhalten. Meist kann die MMU einfach zwischen mehreren Konfigurationen hin- und herwechseln. Betriebssysteme nutzen diese Möglichkeit, indem sie jedem Prozess einen eigenen virtuellen Adressraum zuordnen. Jeder Kontextwechsel geht dann mit dem Wechsel des virtuellen Adressraums einher.

Beschränkung des Zugriffs Paging ordnet jeder Speicherseite individuelle Eigenschaften zu. Diese umfassen beispielsweise:

- Die Beschränkung von Lese- und Schreibzugriffen,
- das Verbot der Codeausführung
- die Festlegung des Speichertyps,
- und das Kontrollieren des Caching-Verhaltens.

Sie ermöglichen es, den Zugriff auf kritische Bereiche wie z.B. den Kernel-Space einzuschränken.

Demand-Paging Der Arbeitsspeicher ist eine knappe Ressource. Gelangt ein System an die Grenzen des physikalischen Speichers, können selten benötigte Seiten auf einen Hintergrundspeicher (HGSP), wie bspw. die Festplatte, ausgelagert werden. Der Inhalt dieser Seiten ist dann nur noch auf dem HGSP vorhanden und muss bei Bedarf später wieder in den Arbeitsspeicher zurück geladen werden. Gegebenenfalls ist dafür das Verdrängen anderer Seiten notwendig.

Vermeidung von Fragmentierung Fragmentierung beschreibt die Verschwendung von Speicherplatz. Es gibt zwei Arten von Fragmentierung, die im folgenden Abschnitt genauer vorgestellt werden.

[1] Beispiele dafür sind die ARM A-Serie, x86-, x86-64 und Alpha-Architektur.
[2] Beispiele dafür sind die ARM M-Serie, PIC, AVR und weitere Mikrocontroller.

2.1.2. Fragmentierung

Um Speicheranfragen variabler Größe zu bedienen muss der zu Verfügung stehende Speicher aufgeteilt werden. Fragmentierung beschreibt die Verschwendung von Speicherplatz, die durch diese Zerstückelung des Speichers entsteht. Sie stellt damit für die Speicherverwaltung ein Problem dar, welches bei der Entwicklung von Algorithmen bedacht werden muss. Fragmentierung tritt neben dem ASP auch auf anderen Speichermedien auf (z.B. Festplatten, Flashspeicher).

Man unterscheidet zwei Arten von Fragmentierung:

Interne Fragmentierung bezeichnet den Verschnitt, der innerhalb eines Speicherfragment entsteht, wenn dieses nicht vollständig genutzt wird.

Externe Fragmentierung bezeichnet die Zerstückelung des Speichers. Also den vergeudeten Speicherplatz zwischen Fragmenten, der aufgrund seiner zu geringen Größe, nicht mehr genutzt werden kann.

Im Falle von Paging entsprechen die Fragmente den Seiten bzw. Kacheln. Eine Speicherverwaltung die rein auf Paging basiert besitzt daher immer einen gewissen Grad an interne Fragmentierung, da die angeforderten Blöcke nur selten der Größe einer Seite entsprechen. Kapitel 4 stellt eine Technik vor, die diese Art der Fragmentierung einzudämmen versucht.

Externe Fragmentierung wird durch Paging effektiv verhindert. Die konstante Seitengröße in Kombination mit einer flexiblen Adressübersetzung erlauben es, auch einen sehr fragmentierten PA noch in einen kontinuierlichen VA abzubilden. Durch eine ungeschickte Verwaltung kann jedoch auch der virtuelle Adressraum fragmentieren (siehe Kapitel 3). Der virtuelle Adressraum kann aufgrund seiner Größe jedoch nicht als knappe Ressource angesehen werden.

Eine gute Speicherverwaltung versucht Fragmentierung soweit möglich zu eliminieren. Es zeigt sich, dass es durchaus sinnvoll sein kann interne Fragmentierung (z.B. aufgrund einer konstanten Blockgröße), zugunsten geringerer externer Fragmentierung, zu tolerieren.

2.1.3. Mathematische Betrachtungsweise

Man kann die Aufgabe der MMU auch als mathematische Funktion f interpretieren:

$$f : X \to Y \times \mathcal{P}(F) \tag{2.1}$$

Eine VPN $v \in X$ wird einer PFN $p \in Y$ zugeordnet und mit Eigenschaften (engl. flags) aus der Menge F versehen. Für die x86-64-Architektur sind die Mengen bspw.

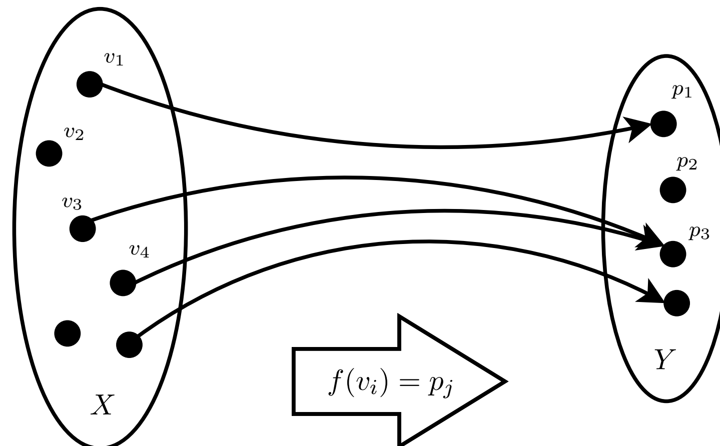


Abbildung 2.2.: Mathematische Ansicht der Memory Management Unit.

wie folgt definiert[3] [4]:

$$X = [0, VA_{max}) \tag{2.2}$$

$$Y = [0, ASP_{max}) \tag{2.3}$$

$$F = \{Present, Read/Write, Accessed, Dirty, Global, ...\} \tag{2.4}$$

Abbildung 2.2 zeigt einige wichtige Eigenschaften der mathematischen Funktion und damit gleichbedeutend von Paging:

- f ist keine injektive Funktion: Eine physikalische Kachel darf durchaus mehreren virtuellen Kacheln zugeordnet sein ($f(v_3) = f(v_4) = p_3$).
- f ist keine surjektive Funktion: Freie Kacheln sind nicht über den VA adressierbar (siehe p_2).
- In der Folge ist f nicht bijektiv und nicht umkehrbar.
- Die Definitionsmenge X darf Lücken enthalten. Der Zugriff auf eine Seite in einer solchen Lücke löst einen Seitenzugriffsfehler (engl. page fault) aus, der vom Betriebssystem (BS) angefangen werden muss.
- $VA_{max} \geq ASP_{max}$: Der VA ist in der Regel größer als der zur Verfügung stehende ASP.

Für die Umsetzung der Abbildungsvorschrift von f gibt es unterschiedliche Ansätze:

[3] Die Variable ASP_{max} steht für den verfügbaren ASP.
[4] Alle verfügbaren Eigenschaften werden im Kapitel 2.2.3 vorgestellt.

- Durch Seitentabellen konstruierte (Such-)Bäume.
- Das Durchsuchen einer inversen Seitentabelle (engl. Inverted Page Table) mit Hilfe einer vorgeschalteten Hashfunktion.
- Durch Software verwaltete Translation Lookaside Buffer (TLB).

Die ersten beiden Varianten werden typischerweise direkt in Hardware implementiert. Für den dritten Ansatz muss das BS einspringen und die Übersetzung in Software übernehmen. Software-TLBs sind daher ein recht flexibler Ansatz.

Im folgenden Kapitel wird das Konzept der Seitentabellen anhand der x86-Architektur genauer vorgestellt. Alternative Paging-Konzepte sowie ein Vergleich gängiger Architekturen befinden sich in Abschnitt 2.5.

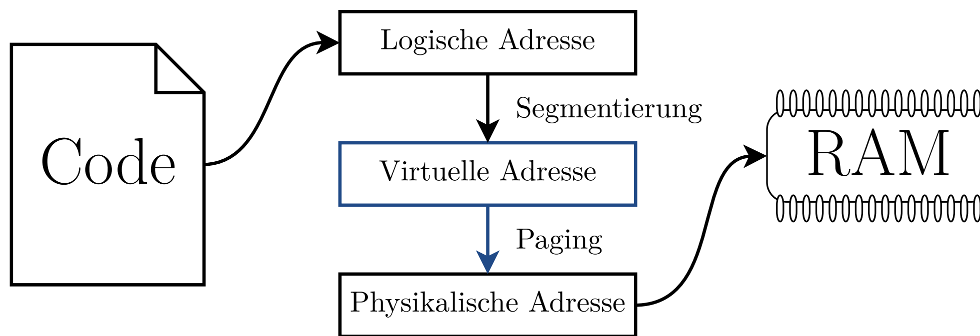


Abbildung 2.3.: Der Weg einer Adresse in der x86-Architektur.

2.2. Die x86-Architektur

Dieses Kapitel stellt das Paging der x86-Architektur vor. Zur x86-Architektur werden alle zu Intels 80386 kompatible Prozessoren (IA-32), sowie die 64-Bit-Erweiterung (IA-32e, bzw. AMDs AMD64) gezählt. Die neuen Prozessoren besitzen mit dem *Longmode* eine neue 64-Bit-Erweiterung der Instruction Set Architecture (ISA). Sie bleiben jedoch weiterhin mit dem 32 Bit *Protected-Mode* sowie dem 16 Bit *Real-Mode* ihrer Vorgänger kompatibel. Eine typische Initialisierung der Kerne beginnt daher im Real-Mode, betritt dann den Protected-Mode und endet schließlich im Longmode. Die Erweiterung der ISA auf 64 Bit hat für das Paging nur geringe Anpassungen erfordert. Auch aufgrund der heutigen Verbreitung, wird daher im Folgenden ein Schwerpunkt auf das Paging im Longmode gelegt.

2.2.1. Historische Entwicklung

Abbildung 2.3 zeigt den Zusammenhang zwischen Paging und Segmentierung. Die Segmentierung (engl. segmentation) ist bereits dem ersten x86-basierten Prozessor Bestandteil der Architektur. Sie bietet ähnlich wie das Paging Funktionen des Speicherschutzes und kann rudimentäre Adressberechnungen ausführen. Aus Kompatibilitätsgründen ist Segmentierung auch in heutigen 64-Bit-Prozessoren noch immer in einer abgespeckten Variante vorhanden. Jedoch besitzt es nun nur noch die Aufgabe die aktuelle Privilegierungstufe (Ring) festzulegen. Adressberechnungen können mit der Segmentierung im Longmode nicht mehr ausgeführt werden, sodass hier die linearen Adressen immer gleich der virtuellen Adressen sind[5].

Die Aufgabe der Adressberechnungen wird in diesem Fall vom Paging übernommen. Damit ist die Segmentierung für diese Arbeit nicht von besonderem Interesse

[5]Diese Nutzung der Segmentierung wird auch als „Flat memory model“ bezeichnet.

	32 Bit Protected-Mode	**64 Bit** Longmode
Stufen	2 (1)	4 (2, 3)
Einträge pro Tabelle	$2^{10} = 1024$	$2^9 = 512$
Größe eines Eintrags	4 B	8 B
Größe einer Seite	4 KiB (4 MiB)	4 KiB (4 MiB, 1 GiB)
Größe des VA	4 GiB	256 TiB
Virtuelle Adresse	32 Bits	48 Bits
Physikalische Adresse	32 (36) Bits	bis 52 Bits
Execute-disable (XD) Flag	nein	ja

Tabelle 2.1.: Vergleich des Pagings zwischen Protected- und Longmode.

und wird nicht näher vorgestellt. Die Tabelle 2.1 zeigt die Paging-relevanten Unterschiede zwischen dem 32-Bit-Protected-Mode und dem 64-Bit-Longmode. Die dort in Klammern angebenen Werte sind alternative Konfigurationen die durch Hugepages erreicht werden können (siehe Abschnitt 2.2.4).

2.2.2. Seitentabellen

Seitentabellen sind die verbreiteste Methode Paging umzusetzen. Sie werden in zahlreichen Architekturen, wie bspw. ARM A-Serie [1], Alpha [3], VAX [5], x86 von Intel, VIA und AMD eingesetzt [10]. Die folgende Beschreibung des Konzeptes ist daher in seinen Grundzügen auch auf andere Architekturen übertragbar.

Wir betrachten zunächst den Aufbau einer virtuellen Adresse in Abbildung 2.5a und 2.6a. Aus ihr wird ersichtlich, dass jede Adresse in mehrere Teile zerfällt:

- Die Indizes #PML4, #PDPT, #PGD und #PGT beschreiben die Positionen von Einträgen innerhalb der Seitentabellen. Gemeinsam bilden sie die VPN.
- Der Offset beschreibt die Position innerhalb einer Kachel. Er wird von der MMU nur durchgereicht und auf die physikalische Adresse der Kachel addiert.

Abbildung 2.4 zeigt den so genannten Pagetable-Walk. Dieser übersetzt eine virtuelle VPN in eine physikalische PFN. Die Indizes der VPN werden genutzt um schrittweise Einträge aus den Seitentabellen auszuwählen. Begonnen wird dabei mit dem PGD im Protected-Mode, bzw. der PML4-Tabelle im Longmode. Die Basisadresse dieser Urtabelle wird dabei durch das Control Register 3 (CR3) festgelegt[6].

[6] Auch diese Adresse zeigt wie alle in den Tabellen genutzten Referenzen auf den physikalischen Speicher.

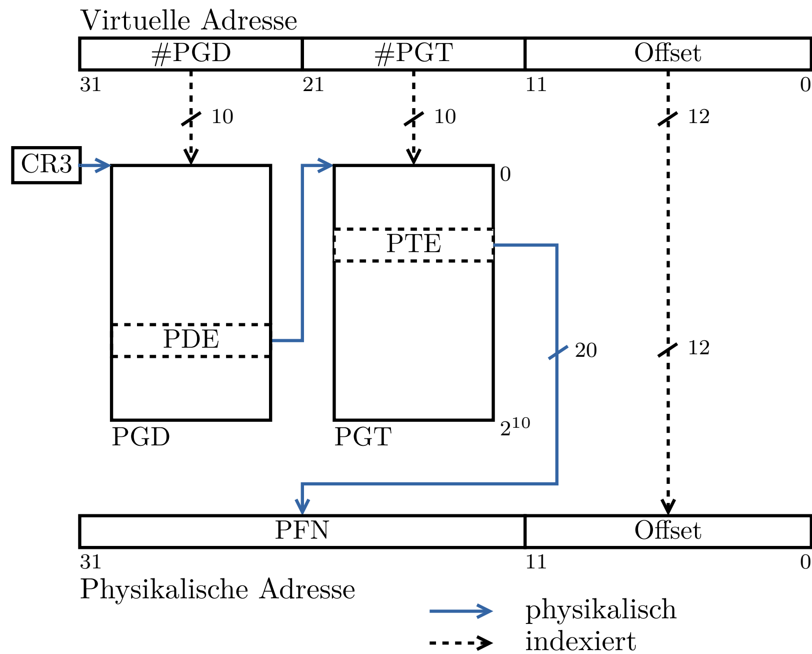


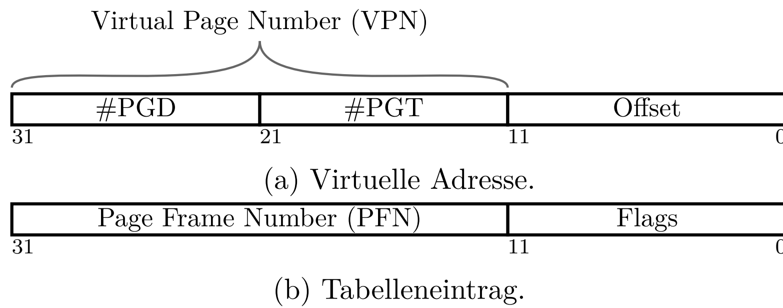
Abbildung 2.4.: Ablauf eines Pagetable-Walks.

Der ausgewählte Eintrag zeigt dann auf die nächste Tabelle. Dieser Vorgang wird wiederholt bis die letzte Tabelle (PGT) erreicht wurde. In diesem Fall enthält der Eintrag die PFN der gesuchten Kachel. Schließlich addiert die MMU den Offset der virtuellen Adresse auf die PFN um die endgültige Adresse zu erhalten.

Trifft der Pagetable-Walk auf einen Eintrag mit nicht gesetztem Present (P) Bit wird dieser abgebrochen. Der Zugriff auf einen nicht abgebildeten Bereich des virtuellen Adressraums löst eine Ausnahme (engl. exception) im System aus. Diese muss vom BS mit Hilfe eines Page-Fault-Handlers abgefangen und behandelt werden. Auch bei der Verletzung von Zugriffsrechten kommt der Page-Fault-Handler zum Einsatz. Typische Anwendungsfälle dafür z.B. sind Demand-Paging oder Copy-on-Write.

In der x86-Architektur ist eine Seitentabelle immer 4 KiB groß. Sie entspricht damit genau der Größe einer Kachel. Mit einer Größe von 4 B pro Eintrag besteht eine Tabelle im Protected-Mode somit immer aus $\frac{4KiB}{4B} = 1024$ Einträgen. Mit der 64 Bit-Erweiterung wurde die Größe eines Eintrags auf 8 B vergrößert. In Folge enthalten die Tabellen nur noch 512 Einträge.

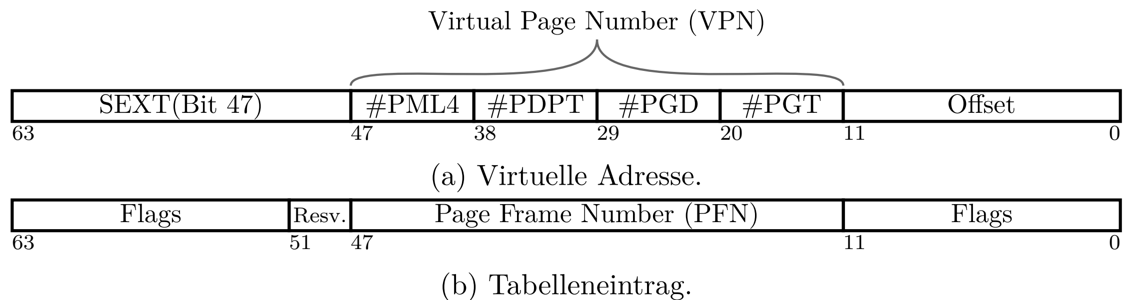
Jeder Tabelleneintrag (vgl. Abbildung 2.5b und 2.6b) besteht aus einer PFN sowie Flags (siehe Abschnitt 2.2.3). Die Referenzen, die während eines Pagetable-Walks aufgelöst werden (PFN), zeigen immer auf physikalische Adressen. Dies ist zwingend nötig, da jeder Zugriff auf eine virtuelle Adresse einen neuen Pagetable-Walk auslösen würde. Die Folge wäre eine endlose Rekursion.



(a) Virtuelle Adresse.

(b) Tabelleneintrag.

Abbildung 2.5.: Virtuelle Adresse und Tabelleneintrag im Protected-Mode.



(a) Virtuelle Adresse.

(b) Tabelleneintrag.

Abbildung 2.6.: Virtuelle Adresse und Tabelleneintrag im Longmode.

2.2.3. Flags

Jeder Eintrag einer Seitentabelle ist mit eine Reihe von Flags versehen. Diese, durch Bits kodierten Eigenschaften, charakterisieren die Zugriffe auf die referenzierte Seite bzw. Tabelle. Tabelle 2.2 zeigt die Flags der x86-Architektur[7]. Ein Pagetable-Walk durchläuft in der Regel mehrere Tabellen. Die Bedeutung der Flags ist dabei über alle Paging-Stufen hinweg fast identisch[8]. Die Flags der dabei besuchten Einträge werden durch die MMU mit Hilfe von logischen Operatoren verknüpft. Das Ergebnis dieser Verknüpfung legt die letztendlich gültigen Eigenschaften der Seite fest. Tabelle 2.2 enthält in der letzten Spalte die genutzten Verknüpfungen.

Das **P** Flag ist das wichtigste Bit jedes Eintrags. Ist es gesetzt verweist der Eintrag auf eine gültige Kachel oder Tabelle. Ansonsten wird der restliche Inhalt des Eintrags ignoriert und hat für die MMU keine Bedeutung.

Die Flags **R/W**, **U/S** und **XD** legen grundlegende Berechtigungen fest. Sie bestimmen ob Inhalte einer Seite gelesen, geschrieben oder ausgeführt werden

[7] Die 64-Bit-Erweiterung ergänzt diese lediglich um das XD Bit (engl. Execution Disabled).

[8] Es gibt feine Unterschiede, die jedoch hier vernachlässigt werden können. Z.B. ist das Global-Flag nur in Einträgen der Page Table (PGT) gültig.

Bit	Name		Verknüpfung
0	P	Present	Und
1	R/W	Read/Write	Und
2	U/S	User/Supervisor	Und
3	PWT	Page-level write-through	
4	PCD	Page-level cache-disable	
5	A	Accessed	
6	D	Dirty	
7	PS	Page Size	
	PAT	Page Attribute Table	
8	G	Global	
63	XD	Execute-disable	Oder

Tabelle 2.2.: Die Flags des Pagings im Longmode.

dürfen. Das U/S Flag kann den Zugriff auf eine Seite nur für die höchste Privilegierungstufe (Ring 0) gestatten. Gewöhnlich ist dies für den Kernel-Space der Fall, um Zugriffe auf Daten des Kernels einzuschränken.

Die Flags **PWT**, **PCD** und **PAT** steuern das Caching-Verhalten der Seiten. So kann z.B. das Caching vollständig abgeschaltet (PCD) oder ein Write-through (PWT) Verhalten erzwungen werden. Mit der Page Attribute Table (PAT) können in Verbindung mit den Memory Type Range Registers (MTRRs) weitere Einstellungen bezüglich des Caching vorgenommen werden.

Die Flags **A** und **D** werden automatisch durch die MMU gesetzt. Sie signalisieren dem BS, dass bestimmte Speicherseiten gelesen bzw. verändert wurden. Diese Informationen können von den Algorithmen des Demand-Pagings genutzt werden um festzustellen welche Seiten schon länger nicht mehr benutzt wurden.

Die Bits 9-11 sowie 52-62 werden von der MMU bei der Adressberechnung ignoriert. Sie können daher genutzt werden um zusätzliche Informationen beispielsweise für das Demand-Paging zu speichern. Die Bits 12-51 enthalten die physikalische Kacheladresse (PFN), die je nach Stufe auf eine weitere Tabelle oder eine Kachel zeigt.

2.2.4. Multilevel-Paging

Sowohl das 32 Bit-Paging als auch die 64 Bit-Erweiterung nutzen mehrstufige Seitentabellen. Im Protected-Mode stehen mit dem PGD und der PGT zwei Tabellen

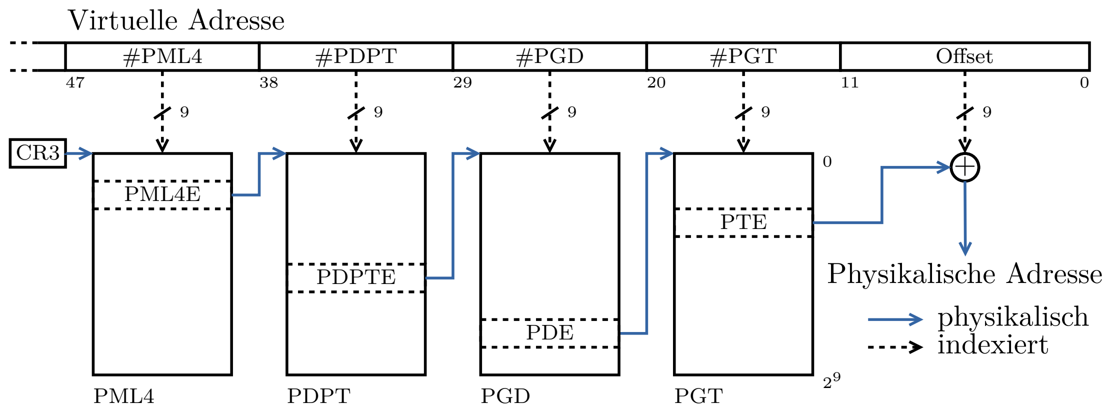


Abbildung 2.7.: Vierstufiges Multilevel-Paging im Longmode.

Stufen	**32 Bit** Protected-Mode		**64 Bit** Longmode	
1	$2^{22}\ B$	$= 4\,\text{MiB}$	$2^{39}\ B$	$= 512\,\text{GiB}$
2	$2^{13}\ B$	$= 8\,\text{KiB}$	$2^{22}\ B$	$= 4\,\text{MiB}$
4	$2^{9}\ B$	$= 512\,\text{B}$	$2^{14}\ B$	$= 16\,\text{KiB}$
8	$2^{7,5}\ B$	$\approx 182\,\text{B}$	$2^{10,5}\ B$	$\approx 1448\,\text{B}$
16	$2^{7,25}\ B$	$\approx 152\,\text{B}$	$2^{9,25}\ B$	$\approx 608\,\text{B}$

Tabelle 2.3.: Größe der Seitentabellen für das Einblenden einer Seite.

zur Verfügung. Im Longmode wurden diese noch um die PML4 und PDPT zu einem vierstufigen System erweitert (siehe Abbildung 2.7).

Der Grund für diese hierarchische Anordnung der Tabellen wird deutlich wenn man den für die Seitentabellen benötigten Speicherplatz betrachtet. Für einen vollständig belegten virtuellen Adressraum eines 32 Bit-Systems sind 4 MiB an Seitentabellen nötig ($4B \cdot \frac{2^{32}}{2^{12}} = 2^{22}B$). Mit der 64 Bit-Erweiterung sind es sogar 512 GiB. Bei einer vollständigen Nutzung des virtuellen Adressraums, ist der benötigte Speicherplatz auch unabhängig von der Anzahl der Paging-Stufen. Erst wenn größere Bereiche des virtuellen Adressraums ungenutzt bleiben werden die Vorteile des mehrstufigen Pagings ersichtlich. Betrachte man dazu den entgegengesetzten Extremfall, dass nur eine einzige Seite des VA belegt ist. In diesem Fall würde man nur eine Seitentabelle pro Stufe benötigen. Tabelle 2.3 zeigt den dazu benötigten Speicherplatz der Tabellen in Abhängigkeit der Stufenanzahl.

Hugepages

Hugepages fassen einen kontinuierlichen Bereich von Kacheln zu einem größeren Speicherbereich zusammen. Trifft ein Pagetable-Walk bereits frühzeitig auf einen

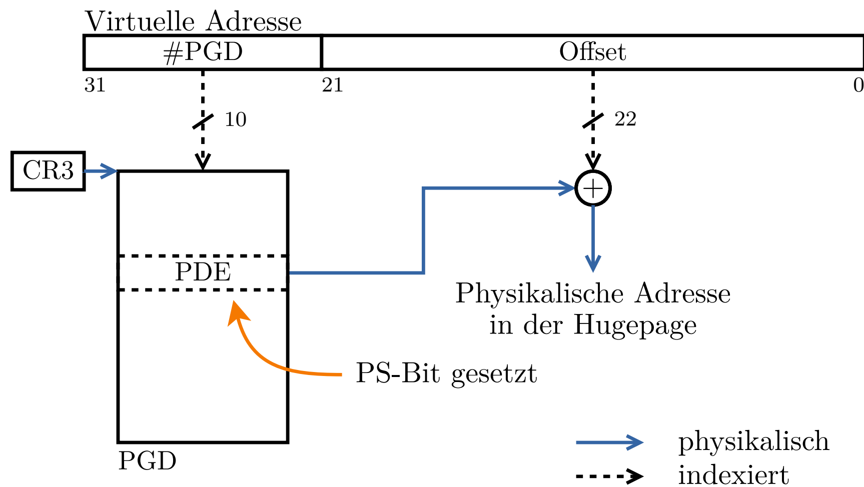


Abbildung 2.8.: 4 MiB Hugepage im Protected-Mode.

Eintrag mit gesetztem PS-Bit stellt dieser eine Hugepage dar. Der Pagetable-Walk wird abgebrochen und verbleibenden Indizes Teil des nun vergrößerten Offsets (siehe Abbildung 2.8).

Im Protected-Mode darf das PS-Bit nur für Einträge der PGD-Tabelle gesetzt sein. Die verbleibenden 10 Bits des PGT-Index vergrößern somit den Offset/Größe der Hugepage auf 22 Bit. Damit sind im Protected-Mode Hugepages mit einer Größe von 4 MiB möglich. Im Longmode darf das PS-Bit zusätzlich auch für Einträge der Page Directory Pointer Table (PDPT) gesetzt sein. Durch die verkürzten Indizes entstehen so Offsets von 21 Bits (PGD) und 30 Bits (PDPT), die Hugepages der Größe 2 MiB und 1 GiB entsprechen.

Hugepages haben den Vorteil, dass keine Seitentabellen für die unteren Stufen benötigt werden. Dies reduziert den Speicherverbrauch und entlastet gleichzeitig den TLB (siehe Abschnitt 2.2.6). Durch den verkürzten Pagetable-Walk kann dieser schneller ausgeführt werden. Es ist jedoch zu beachten, das 1 GiB-Hugepages nur von neueren Mikroarchitekturen ab Intels Westmere vorhanden sind. Weiterhin müssen Hugepages explizit über das PSE Bit des Control Register 4 (CR4) aktiviert werden.

2.2.5. Kanonische Adressen

Betrachtet man die virtuelle Adresse in Abbildung 2.6a wird ersichtlich, dass von den zu Verfügung stehenden 64 Bits lediglich 48 Bits genutzt werden. Der VA kann also derzeit maximal 256 TiB der theoretisch möglichen 16 EiB umfassen. Da dies jedoch für aktuelle Systeme vollkommen ausreichen ist, hat man sich entschieden

zugunsten der Hardwarekosten auf die verbleibenden 16 Bits zu verzichten[9].

Jede Adresse der x86-64-Architektur muss dem „kanonischen Adressformat“ entsprechen. Dieses besagt, dass die Bits 48 bis 64 der Adresse gleich dem 47. Bit der Adresse entsprechen müssen. Eine Verletzung dieser Vorschrift löst einen General-Protection-Fault aus. Diese Vorzeichenerweiterung (engl. sign extension) teilt den 256 TiB großen Adressraum in zwei Bereiche à 128 TiB auf (siehe Abbildung 2.9). Die Adressen des oberen Bereichs können als Zweierkomplement Darstellung aufgefasst werden. In diesem Fall erstreckt sich der virtuelle Adressraum dann über den folgenden Bereich: $[-2^{48}, 2^{48})$. Mit Hilfe des arithmetischen Schiebeoperators kann die Vorzeichenerweiterung einfach berechnet werden (siehe Quellcode 2.1). Aufgrund des vorzeichenbehafteten Datentyps `ssize_t addr` wird eine arithmetische Rechtsverschiebung vorgenommen, die das Vorzeichen beibehält.

```
1 size_t sign_extend(ssize_t addr, int bits) {
2   int shift = BITS - bits; // 64 - 48 = 16
3   return (addr << shift) >> shift;
4 }
```

Listing 2.1: Berechnung der Vorzeichenerweiterung.

Da das kanonische Adressformat nur im Longmode Anwendung findet, benutzt MetalSVM das folgende Makro:

```
1 #ifdef CONFIG_x86_32
2  #define CANONICAL(addr)  (addr)
3 #elif defined(CONFIG_X86_64)
4  #define CANONICAL(addr)  sign_extend(addr, 48)
5 #endif
```

2.2.6. Translation Lookaside Buffer

Um Speicherzugriffe zu beschleunigen werden die Ergebnisse der Pagetable-Walks im sogenannten Translation Lookaside Buffer zwischengespeichert. Dieser beinhaltet sowohl die PFN also auch die für die Seiten geltenden Flags. Jeder Speicherzugriff löst zuerst einem Lookup im TLB aus. Ist das Ergebnis noch im Cache vorhanden (Hit) kann dieses direkt genutzt werden. Andernfalls (Miss) muss es

[9] Die x86-64 ISA bietet trotzdem Unterstützung für 64-Bit-Adressierungen, sodass in Zukunft abwärtskompatible Systeme mit einem 64 Bit großen virtuellem Adressraum realisierbar wären.

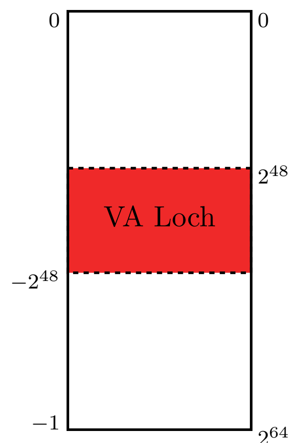


Abbildung 2.9.: Das Loch des virtuellen Adressraums.

erst durch einen erneuten Pagetable-Walk bestimmt werden.

Der TLB ist ein schneller Puffer, der als Assoziativspeicher auf in allen Kernel der CPU vorhanden ist. Im Vergleich zu L1/L2-Caches wird der TLB aber nicht kohärent gehalten. Durch die Sicherstellung von Cache-Kohärenz wird bei Mehrprozessorsystemen mit mehreren Caches verhindert, dass die einzelnen Caches für dieselbe Speicheradresse inkonsistente Daten enthalten. Daher rührt auch die begriffliche Abgrenzung des Puffers zum Cache. Das bedeutet, dass die im Folgenden beschriebene Invalidierung des TLB vom BS übernommen werden muss.

Vergleicht man den TLB mit den Caches ist jedoch auch eine Gemeinsamkeit zu finden: Beide Zwischenspeicher profitieren von dem Lokalitätsprinzip der Daten und Instruktionen. Aus diesem Grund werden in der Regel auch zwei getrennte TLB für Daten- und Instruktionszugriffe eingesetzt. Modernere Mikroarchitekturen, seit Intels Nehalem, besitzen zudem noch einen gemeinsam genutzten L2-TLB. Die Tabelle 2.4 zeigt die Organisation des TLBs für die aktuelle Haswell Prozessoren.

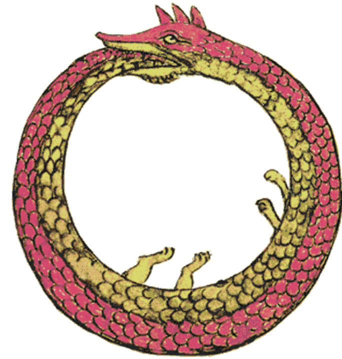
Invalidierung

Ändert das BS Einträge der Seitentabellen müssen eventuell im TLB vorgehaltene Einträge invalidiert werden. Auch ein Kontextwechsel, also das Austauschen aller Seitentabellen über das CR3, führt zu einer Invalidierung des gesamten TLB. Um die Kosten dieses so genannten TLB-Flushes zu minimieren besitzt die x86-Architektur „globale Seiten“. TLB-Einträge mit gesetzten Global-Flag werden von Invalidierung durch einen Kontextwechsel ausgeschlossen. Sie eignen sich damit für Speicherbereiche, die von allen Prozessen gemeinsam genutzt werden. Ein Beispiel hierfür ist der Kernel.

Level	Seitengröße	Einträge	Assoziativität
Instruktionen	4 KiB	128	4-fach
Instruktionen	2 MiB/4 MiB	8 / Thread	4-fach
L1 Daten	4 KiB	64	4-fach
L1 Daten	2 MiB/4 MiB	32	4-fach
L1 Daten	1 GiB	4	4-fach
L2 gemeinsam	4 KiB, 2 MiB/4 MiB	1024	8-fach

Tabelle 2.4.: TLB-Parameter der Haswell Mikroarchitektur von Intel [9], [11].

2.3. Selbstreferenzialität



Der Ouroboros, ein Drache der sich fortlaufend selbst verzehrt, als Symbol für die Selbstreferenzierung[1].

Die Implementierung des Pagings ist Teil der Speicherverwaltung und wird damit im Kernel ausgeführt. In dessen Kontext ist wie im User-Space das Paging immer aktiviert. Während es im Protected-Mode noch leicht deaktiviert werden konnte, ist dies für den Longmode nicht mehr möglich. Anders ausgedrückt ist der Longmode nur dann aktiv, wenn auch das Paging aktiviert ist[10].

Folglich sind Speicherzugriffe nur auf den virtuellen Adressraum möglich. Die Referenzen auf Seitentabellen im CR3-Register, sowie die PFNs in den Einträgen, enthalten jedoch physikalische Adressen. Zur Verwaltung der Seitentabellen müssen daher die physikalischen Adressen in Virtuelle übersetzt werden. Nur so kann der Kernel dem Lauf einen Pagetable-Walks folgen. Dies entspricht quasi dem Gegenstück der in Abschnitt 2.1.3 vorgestellten Abbildungsfunktion der MMU.

Ursprünglich wurde das Paging von MetalSVM für den 32-Bit-Protected-Mode entwickelt. Damit besaß es nur 2 Stufen und maximal $2^{10} = 1024$ Seitentabellen (PGTs). Diese wurden mit Hilfe eines `pgt_container` in den virtuellen Adressraum des Kernel-Space eingeblendet. Dieser Container ist selbst eine Page Table (PGT) und wird in das PGD des aktuellen Prozesses eingetragen. Seine Einträge zeigen jedoch nicht auf Page Frames (PFs), sondern auf andere Seitentabellen (siehe Abbildung D.1). Auf diese Weise kann auf alle Seitentabellen direkt zugegriffen werden.

Mit der Einführung des Longmodes ist dieses Vorgehen nicht mehr praktikabel. Das vierstufige Paging erlaubt bis zu $2^{3 \cdot 9} + 1 = 134217729$ Seitentabellen. Ein einziger Container würde nicht mehr genügen. Für alle Tabellen der höheren Stufen müssten eine solche virtuelle Kopie verwaltet werden.

Mit dem neuen Ansatz geht MetalSVM einen anderen Weg. Die Seitentabellen werden nicht mehr einzeln und explizit im Kernel-Space eingeblendet. Stattdessen zeigt ein Eintrag der Urtabelle (PGD bzw. PML4) auf die Urtabelle selbst (siehe roter Pfeil in Abbildung 2.11). Dies führt dazu, dass der virtuelle Adressraum,

[1] `http://en.wikipedia.org/wiki/File:Ouroboros.png`

[10] Genaueres steuern die Bits PG, PE des Control Register 0 (CR0) bzw. LMA, LME des IA-32 Extended Feature Enable Register (EFER).

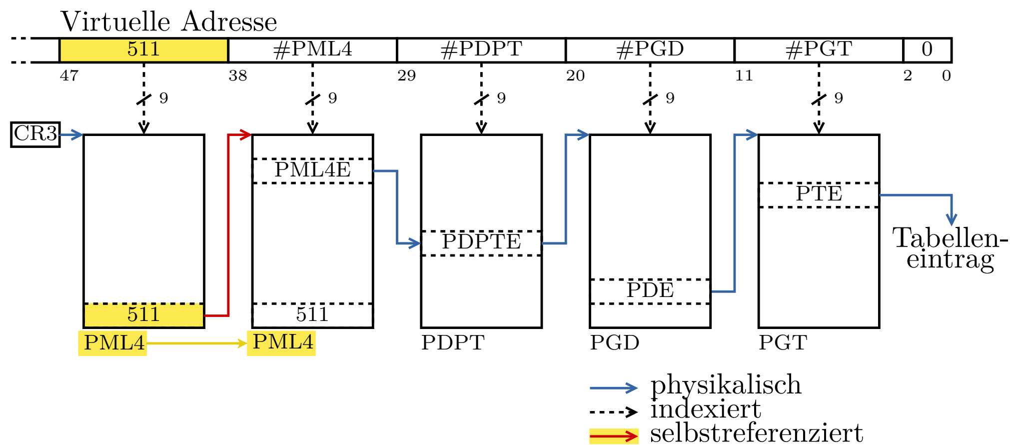


Abbildung 2.11.: Longmode-Paging mit selbstreferenzierender PML4-Tabelle.

der durch diesen Eintrag abgebildet wird, auf die zurzeit aktiven Seitentabellen zeigt. Ein Eintrag der Stufe i (z.B. PDPT) wird durch die Selbstreferenzierung als Eintrag der Stufe $i + 1$ (PML4) interpretiert. Führt man diesen Pagetable-Walk fort erhält man schließlich die Adresse einer Page Table (PGT) (und nicht mehr einer Kachel). Mit den obersten 9 Bits des Offsets kann dann ein Eintrag der PGT ausgewählt werden. Man vergleiche dazu auch den Ablauf eines normalen Pagetable-Walks in Abbildung 2.7.

MetalSVM nutzt den letzten Eintrag der Page Map Level 4 (PML4) Tabelle für die Selbstreferenzierung. Diese Wahl dieses Eintrags ist quasi willkürlich. Sie hat nur den Vorteil, dass die Tabellen am Ende des VA eingeblendet werden und dort nicht mit dem User-Space oder Kernel-Space kollidieren. Durch die Selbstreferenzierung werden die Seitentabellen linear im virtuellen Adressraum eingeblendet.

2.3.1. Beispiel

Das folgende Beispiel soll den Zugriff auf die Seitentabellen mit Hilfe der Selbstreferenzierung weiter verdeutlichen. Wir wählen die Adresse A mit $A =$ `0x1234567890AB`. Gesucht ist die Adresse T des ihr zugehörigen Eintrags der Seitentabelle (PGT). Die Adresse A lässt sich wie folgt in ihre Bestandteile (Indizes und Offset) zerlegen:

Index	Formel	Wert
$\#PML4_A$	`(A » 39) & 0x1FF`	`0x024`
$\#PDPT_A$	`(A » 30) & 0x1FF`	`0x0D1`
$\#PGD_A$	`(A » 21) & 0x1FF`	`0x0B3`
$\#PGT_A$	`(A » 9) & 0x1FF`	`0x189`
$Offset_A$	`(A » 0) & 0xFFF`	`0x0AB`

Tabelle	32 Bit Protected-Mode	64 Bit Longmode
PGT	0xFFC00000	0xFFFFFF8000000000
PGD	0xFFFFF000	0xFFFFFFFFC0000000
PDPT	–	0xFFFFFFFFFFE00000
PML4	–	0xFFFFFFFFFFFFF000

Tabelle 2.5.: Basisadressen der Seitentabellen im virtuellen Adressraum.

Durch die Selbstreferenzierung zeigt der letzte Eintrag der PML4-Tabelle auf die Tabelle selbst. Das führt dazu, dass der PDPT-Index wiederum einen Eintrag aus derselben PML4-Tabelle auswählt (siehe Abbildung 2.11). Auch der folgende PGD-Index wählt einen Eintrag aus der nun referenzierten PDPT. Gleiches gilt für den PGT-Index, der einen Eintrag aus dem PGD auswählt. Dieser Eintrag des PGD zeigt jedoch auf eine PGT und nicht wie üblich auf ein PF. Mit Hilfe des Offsets können wir also direkt auf die Seitentabelle zugreifen und deren Einträge ändern. Alle Indizes werden quasi um eine Stufe verschoben (siehe Abbildung 2.12). Wie wählen daher $\#PDPT_T = \#PML4_A$, $\#PGD_T = \#PDPT_A$, $\#PGT_T = \#PGD_A$:

Index	Formel	Wert
$\#PML4_T$	511	0x1FF
$\#PDPT_T$	$\#PML4_A$	0x024
$\#PGD_T$	$\#PDPT_A$	0x0D1
$\#PGT_T$	$\#PGD_A$	0x0B3
$Offset_T$	$\#PGT_A \cdot 8$	0xC48

Kombiniert man diese Indizes erhält man die Adresse $T =$ 0xFFFFFF80002B3C48. Sie zeigt direkt auf den Eintrag der Seitentabelle (PGT) der Adresse A.

Das gerade besprochene Beispiel zeigte die Berechnung der Adresse eines Eintrags der Page Table (PGT). Die Selbstreferenzierung ermöglicht es jedoch auch auf die Einträge höherer Tabellen (PML4, PDPT, PGD) zuzugreifen. Dazu ist es nötig mehrfach die selbstreferenzierte Tabelle zu indexieren. Das heißt, dass auch der PDPT-Index wiederum die PML4 Tabelle auswählt ($\#PDPT_T = 511$). Führt man dieses Vorgehen fort ($\#PML4_T = \#PDPT_T = \#PGD_T = \#PGT_T = 511$) stößt man so schließlich auf die PML4-Tabelle selber. Diese wiederholte Nutzung der Selbstreferenzierung äußert sich in einem wiederholten verschieben der Indizes (siehe Abbildung 2.12). Tabelle 2.5 zeigt die daraus resultierenden Adressen der Seitentabellen.

Der folgende Quelltext zeigt die Berechnung der Adressen wie sie am Beispiel erläutert wurde. Zeile 2 verschiebt alle Indizes in die nächst niedrigere Stufe. In der darauf folgenden Zeile wird die Basisadresse aller Seitentabellen (`PAGE_MAP_PGT`) addiert. Das Ergebnis dieser beiden Zeilen ist die Adresse des PGT-Eintrags (`level` ↪ `= 0`).

Wird der Eintrag einer höheren Tabelle gesucht, müssen die Indizes weiter verschoben werden (Zeile 5). Durch den vorzeichenbehafteten Typ der Addresse (`ssize_t`) wird hier eine arithmetische Verschiebung vorgenommen. Diese sorgt dafür, dass der Index der PML4-Tabelle weiterhin den Wert 511 behält. Der Rest der Funktion richtet die Adresse noch an der Größe eines Tabelleneintrags aus und stellt sicher, dass die zurückgegebene Adresse auch dem kanonischen Format (Abschnitt 2.2.5) entspricht.

```
1 page_entry_t* virt_to_entry(ssize_t addr, int level) {
2   addr >>= PAGE_MAP_BITS; // get offset in PGT table
3   addr |= PAGE_MAP_PGT; // add PGT base
4
5   addr >>= level * PAGE_MAP_BITS;
6   addr &= ~(sizeof(size_t) - 1); // align to page_entry_t
7
8   return (page_entry_t*) CANONICAL(addr);
9 }
```

Listing 2.2: Berechnung der Adresse eines Tabelleneintrags.

Der entgegengesetzte Weg unterscheidet sich lediglich durch die Richtung der Schiebeoperation.

```
1 size_t entry_to_virt(page_entry_t* entry, int level) {
2   size_t addr = (size_t) entry;
3
4   addr <<= (level+1) * PAGE_MAP_BITS;
5
6   return CANONICAL(addr);
7 }
```

Listing 2.3: Berechnung der Adresse zu einem Tabelleneintrag.

Tabellen die vielleicht noch gar nicht existieren (z.B. ein Loch im VA) besitzen einen Platz im VA. Ein Zugriff auf eine nicht-existierende Tabelle löst wie bei den Seiten einen Page-Fault aus, der genutzt werden kann um die Tabelle zu erzeugen.

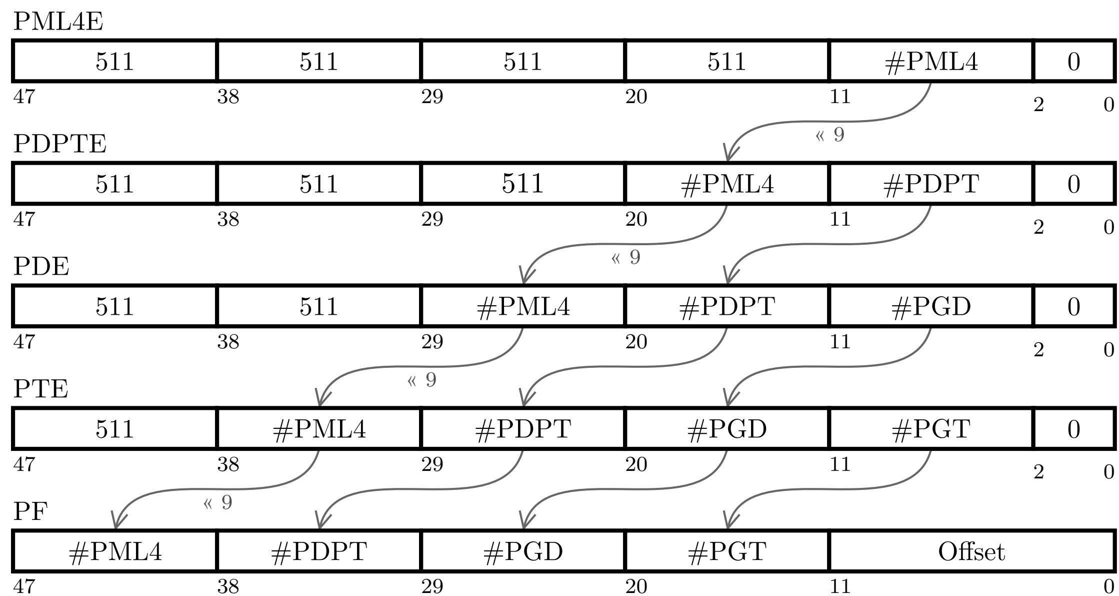


Abbildung 2.12.: Adressen der Seitentabelleneinträge durch Selbstreferenzierung.

Die Technik der selbstreferenzierenden Seitentabellen wir im Englischen auch als „self-mapped“ oder „recursive mapping“ bezeichnet. Dieser Trick ist auf bestimmte Eigenschaften der Seitentabellen angewiesen. Die Tabelleneinträge müssen über alle Stufen hinweg ähnlich aufgebaut sein. Hätten die in diesem Beispiel verwendeten Einträge (PDPT und PML4) ein unterschiedliches Format, wäre ein korrekter Pagetable-Walk nicht möglich. Folglich müssen alle Tabellen auch die gleich Anzahl an Einträgen besitzen.

Aus Tabelle 2.7 wird ersichtlich, dass diese Voraussetzungen auch auf das Paging im Protected-Mode zutreffen. Auch für ARMs neue ARMv8-A (AArch64) Architektur oder die Alpha-Architektur wäre ein Einsatz denkbar. Architekturen wie MIPS, die einen Software-TLB einsetzen, können ein Intel 80386 kompatible Architektur (x86)-ähnliches Paging per Software implementieren, und so von diesem Trick profitieren.

2.3.2. Vorteile

Der Einsatz von selbstreferenzierende Seitentabellen hat sowohl Vor- als auch Nachteile, die im Folgenden kurz angerissen werden sollen.

- Sie ermöglichen es komplexe und mehrstufige Paging-Architekturen mir recht einfachem und eleganten Code zu implementieren. So kann für das Durchlaufen der Seitentabellen eine rekursive Funktion genutzt werden. Jeder Funktionsaufruf durchläuft dabei eine einzige Tabelle und ruft sich für jeden vor-

handen wiederum Eintrag selbst auf. Nach dem die Tabelle vollständig durchlaufen wurde kehrt sie zur übergeordneten Tabelle zurück.

- Auch der Einsatz des Page-Fault-Handlers zum Erstellen nicht existierender Seitentabellen wäre möglich. Die hier vorgestellte Implementierung nutzt jedoch den zuvor vorgestellten Ansatz mit rekursiven Funktionsaufrufen. Durch die Selbstreferenzierung besitzt jede Tabelle eine eindeutige Adresse im VA auch wenn diese Tabelle gar nicht existiert. In diesem Fall würde ein Zugriff auf diese Tabelle einen Seitenfehler (engl. page fault) auslösen, der diese Tabelle erzeugen müsste. Die neu erzeugte Tabelle müsste dann durch die übergeordnete Tabelle referenziert werden. Auch diese kann wiederum nicht existieren und damit einen verschachtelten Page-Fault auslösen.

- Andere BS blenden Seitentabellen aller Prozesse im Kernel-Space ein. Für den gesamten VA eines Prozesses wären dazu 4 MiB bzw. 512 GiB an Seitentabellen (32- bzw. 64 Bit) nötig. In der Regel besitzt der VA eines Prozesses jedoch große Löcher (engl. sparse VA). Für diese Löcher müssen dank mehrstufigem Paging keine Seitentabellen vorgehalten werden. Ohne Selbstreferenzierung muss das BS den virtuellen Speicher für diese Tabellen im Kernel-Space selbst verwalten.

- Die *virtuellen Kopien* der Seitentabellen entfallen. Durch die Selbstreferenzierung werden die Fähigkeiten der MMU genutzt. Diese übernimmt die Berechnung der Adressen der Seitentabellen.

- Durch geschicktes Setzen der Paging-Flags können diese Ihre Wirkung auch auf die Tabellen selbst entfalten. So können bspw. die PGTs (nicht ihre Einträge) mit den Global (G) Flag versehen werden. Auf diese Weise können TLB Einträge der Kernel-Space-Seitentabellen über Kontextwechsel hinweg beibehalten werden.

2.3.3. Nachteile

- Selbstreferenzierende Seitentabellen können nicht auf jeder Pagingarchitektur eingesetzt werden, da sie bestimmte Voraussetzungen an die Struktur der Tabellen stellen. Die Tabellen aller Paging-Stufen müssen alle gleich viele Einträge umfassen. Die Einträge dieser Tabellen müssen den gleichen Aufbau besitzen. Ein Eintrag aus der PML4 Tabelle muss auch als Eintrag der PDPT Tabelle fungieren können.

- Durch die Selbstreferenzierung sind standardmäßig nur die Tabellen des aktuellen Prozesses sichtbar. Um auch die Tabellen anderer Prozesse bearbeiten

zu können, bspw. für das Forken von Prozessen, ist daher eine kleine Zwischenlösung nötig (siehe Abschnitt 2.4.6).

- Auf den ersten Blick ist dieser Ansatz nicht sehr intuitiv. Auch wenn der Code schön kompakt und elegant sein mag, fördert dies nicht unbedingt dessen Wartbarkeit und Nachvollziehbarkeit.
- Ein gewisser Anteil des virtuellen Adressraums wird für die Seitentabellen benötigt und steht den Anwendungen nicht zur Verfügung:

32 Bit $\frac{1}{2^{10}} \approx 0,1\% => \frac{1}{1024} \cdot 2^{32}\ B = 4\ MB$

64 Bit $\frac{1}{2^{9}} \approx 0,2\% => \frac{1}{512} \cdot 2^{48}\ B = 512\ GB$

2.4. Implementierung

MetalSVM ist als BS für die x86-Architektur entworfen worden. Auch an einer Portierung auf die ARM-Architektur wurde gearbeitet. Im Rahmen dieser Arbeit wurde das Paging mit einem alternativen Ansatz neu implementiert.

Ziel war es eine möglichst einfache und minimalistische Implementierung zu finden. Soweit möglich wurde der Code plattformunabhängig und kompakt gehalten. Auf erweiterte Funktionen wie Demand-Paging, das Auslagern von Seiten, Hugepages oder Copy-on-Access wurde daher verzichtet.

Der neue Ansatz nutzt selbstreferenzierende Seitentabellen. Diese werden auch von Microsofts NT-Kernel eingesetzt [4]. Dieser Ansatz ermöglicht es, die gleiche Codebasis sowohl für das zweistufige Paging im Protected-Mode als auch für das vierstufige Paging des Longmode zu verwenden.

2.4.1. Operationen auf Seitentabellen

Die Hauptaufgabe der Paging-Implementierung ist es Operationen auf den Seitentabellen auszuführen. Zu diesen gehören:

`map_region()` Das Einblenden von physikalischen Kacheln in den virtuellen Adressraum (siehe Listing A.1).

`unmap_region()` Das Löschen einer Zuordnung.

`copy_page_map()` Das Kopieren aller Seitentabellen für einen neuen Prozess (siehe Listing A.2).

`drop_page_map()` Das Freigeben aller Seitentabellen eines Prozesses.

`set_page_flags()` Das Setzen von Flags für einen Speicherbereich.

Die Operationen arbeiten dabei immer auf einem kontinuierlichen Bereich des virtuellen Adressraums. Mit der in Kapitel 2.4.2 vorgestellten Interpretation der Seitentabellen, entsprechen diese Bereiche jeweils einem Teilbaum (siehe Abbildung 2.13). Die Operationen werden dabei wiederholt auf jeden Knoten dieses Teilbaums angewendet. Jeder Knoten des Baums entspricht einem Eintrag in einer Seitentabelle. Die Wurzel symbolisiert das CR3; die Blätter Einträge in den PGTs.

Die von Bäumen bekannte Tiefensuche (engl. depth-first search, DFS) wird genutzt um einen Baumdurchlauf (engl. tree traversal) durchzuführen. Dieser Durchlauf legt die Reihenfolge fest in welcher die Operationen auf die Tabelleneinträge angewendet werden.

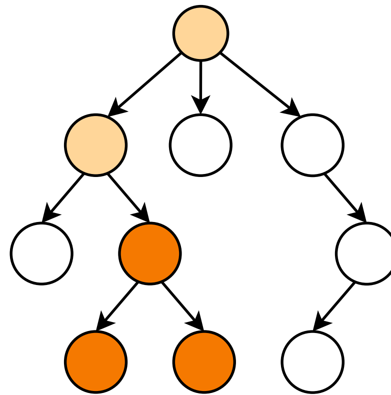


Abbildung 2.13.: Ein orange hervorgehobener Teilbaum.

2.4.2. Seitentabellen als Baum

Die Seitentabellen lassen sich als geordneter Baum interpretieren. In einem geordneten Baum sind die Kinder jedes Knoten nach einem bestimmten Kriterium geordnet. Im Falle der Einträge ist dies ihre Position innerhalb der jeweiligen Seitentabelle.

Dieser Baum lässt sich wie folgt beschreiben:

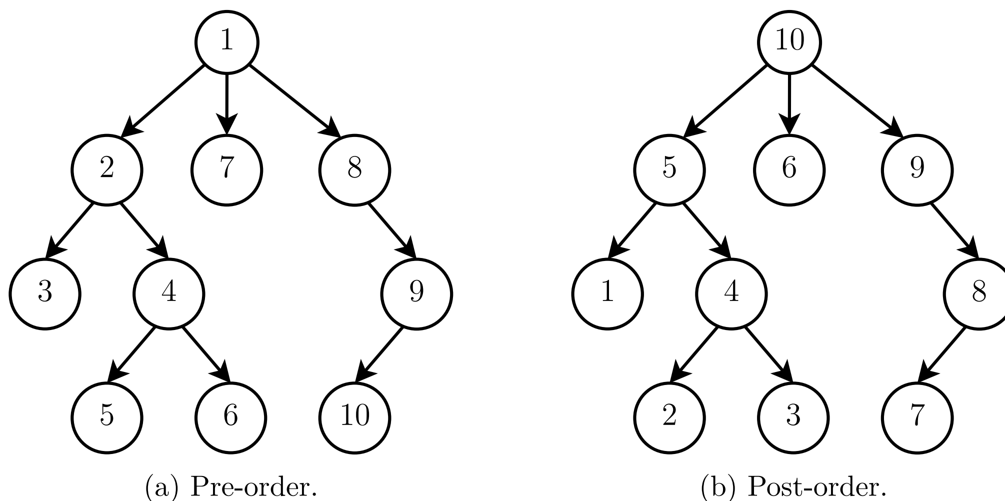
- Der Wurzelknoten symbolisiert das Page Directory (Protected-Mode) bzw. PML4 (Longmode).
- Die Blätter des Baums stellen die Seiten dar.
- Die Höhe entspricht der Anzahl der Paging-Stufen + 1.
- Der Verzweigungsgrad (engl. branching factor) ist gleich der Anzahl der Einträge pro Seitentabelle.
- Bereiche des virtuellen Adressraums entsprechen einem Teilbaum.

2.4.3. Durchläufe durch Seitentabellen

Abbildung 2.14 zeigt zwei Möglichkeiten einen Baum zu durchlaufen. Sie unterscheiden sich dabei lediglich in der Reihenfolge in welcher die Wurzel und Kind-Knoten besucht werden. Die Teilbäume der Kind-Knoten werden dabei rekursiv mit der gleichen Vorschrift durchlaufen.

Pre-order Durchlauf

1. Besuche den Wurzel-Knoten.



(a) Pre-order. (b) Post-order.

Abbildung 2.14.: Baumdurchläufe.

2. Durchlaufe alle Kind-Knoten rekursiv mit einer Pre-order Durchlauf.

Post-order Durchlauf

1. Durchlaufe alle Kind-Knoten rekursiv mit einer Post-order Durchlauf.
2. Besuche den Wurzel-Knoten.

Für die hier vorgestellten Durchläufe können auf beliebige geordnete Bäume, nicht ausschließlich Binärbäume, angewendet werden. Die Bäume (Seitentabellen) besitzen 512 (bzw. 1024) Kind-Knoten, die geordnet nach ihrer Position innerhalb der Seitentabelle durchlaufen werden. Ein sogenannter In-order Durchlauf kann nur auf Binär-Bäume sinnvoll angewendet werden und findet daher bei dem Durchlaufen der Seitentabellen keine Anwendung.

Die verschiedenen Operationen erfordern unterschiedliche Durchläufe durch die Seitentabellen. So ist für das Anlegen einer Zuordnung (`map_region()`) ein Pre-order Durchlauf nötig. Bevor Einträge in der PGT geändert werden können, muss sichergestellt werden, dass die übergeordneten Seitentabellen (PGD, PDPT und PML4) existieren. Für das Löschen von Zuordnungen (`unmap_region()`) ist im Gegensatz ein Post-order Durchlauf nötig. Seitentabellen (Teilbäume) dürfen erst gelöscht werden, wenn alle Kindknoten zuvor entfernt wurden.

2.4.4. MetalSVM

Der folgende Abschnitt stellt einige wichtige Funktionen der Implementierung[11] vor. Die folgenden Quelltexte wurden aus Gründen der Leserlichkeit vereinfacht. Anweisungen zur Synchronisierung (engl. spinlocks) und Fehlerbehandlung (engl. assertions) wurden entfernt.

Auszug 2.4 zeigt Konstanten für das Paging im Longmode (vgl. dazu Tabellen 2.5 und 2.1).

```
#define PAGE_BITS         12
#define BITS              64
#define VIRT_BITS         48
#define PHYS_BITS         52
#define PAGE_MAP_BITS     9
#define PAGE_MAP_LEVELS   4
#define PAGE_MASK         0x000FFFFFFFFFF000

#define PAGE_SIZE         (1 << PAGE_BITS)
#define PAGE_MAP_ENTRIES  (1 << PAGE_MAP_BITS)

#define PAGE_MAP_PML4     0xFFFFFFFFFFFFF000
#define PAGE_MAP_PDPT     0xFFFFFFFFFFE00000
#define PAGE_MAP_PGD      0xFFFFFFFFC0000000
#define PAGE_MAP_PGT      0xFFFFFF8000000000

#define PG_PRESENT       (1 << 0)
#define PG_RW            (1 << 1)

  [...]
```

Listing 2.4: Konstanten des Paging im Longmode.

MetalSVM blendet die Seitentabellen, mit Hilfe der zuvor beschriebenen Selbstreferenzierung, am Ende des virtuellen Adressraums ein. Die Konstante in Listing 2.5 enthällt die Adresse der höchstwertigen Tabelle (PML4 bzw. PGD). Diese Konstante repräsentiert damit die Wurzel des Baums der Seitentabellen und wird im Folgenden sie als Ausgangspunkt für die Baumdurchläufe dienen.

```
page_entry_t* const current_map = (page_entry_t*) (1*PAGE_MAP_PML4);
```

Listing 2.5: Adresse der PML4-Tabelle.

[11] Für weiteres siehe: `metalsvm/arch/x86/mm/` und `metalsvm/arch/x86/include/asm/page*.h`.

Um sich frei im Baum der Seitentabellen bewegen zu können ist es notwendig die Adressen von Kind- und Elternknoten zu kennen. Mit den in Kapitel 2.3 beschriebenen Routinen lassen sich diese Adressen über die folgenden Funktionen berechnen. Die Funktion `get_child_entry()` liefert dabei immer die Adresse der nächst niedrigeren Tabelle zurück. Zum Beispiel die Adresse einer PGD-Tabelle zu einem gegebenen PDPT-Eintrag. Die Funktion `get_parent_entry()` ist das Gegenstück zu dieser Funktion. Sie ermöglich das Zurückkehren aus einer untergeordneten- zur Elterntabelle. Beide Funktionen geben einen Zeiger von Typ `page_entry_t` zurück. Dieser Typ ist genauso groß wie ein Eintrag der Seitentabellen (4 B bzw. 8 B). Im Falle der Funktion `get_child_entry()` zeigt der Zeiger auf den ersten Eintrag der Kind-Tabelle. Er kann damit wie ein Array indexiert werden.

```
page_entry_t* get_child_entry(page_entry_t *entry) {
  size_t child = (size_t) entry;

  child <<= PAGE_MAP_BITS;

  return (page_entry_t*) CANONICAL(child);
}

page_entry_t* get_parent_entry(page_entry_t *entry) {
  ssize_t parent = (size_t) entry;

  parent >>= PAGE_MAP_BITS;
  parent |= PAGE_MAP_PGT;
  parent &= ~(sizeof(size_t) - 1); // align to page_entry_t

  return (page_entry_t*) CANONICAL(parent);
}
```

Das Listing 2.6 zeigt die Berechnung der physikalischen Adresse zu einer gebenen virtuellen Adresse. Hierzu wird zunächst in Zeile 2 der zugehörige Eintrag der Seitentabelle bestimmt. Anschließend wird der Offset der virtuellen Adresse auf die physikalischen Kacheladresse addiert (Zeile 7).

```
size_t virt_to_phys(size_t addr) {
  page_entry_t* entry = virt_to_entry(addr, 0); // get the
      ↪ PGT entry

  size_t off =   addr & ~PAGE_MASK; // offset within page
  size_t phy = *entry &  PAGE_MASK; // physical page frame
```

```
    ↪ number
6
7   return phy | off;
8 }
```

Listing 2.6: Berechnung einer physikalischen Adresse.

2.4.5. Beispiel eines einfachen Durchlaufes

Das Listing 2.7 zeigt einen Durchlauf durch die Seitentabellen, der das Auftreten bestimmter Flags in den Tabellen zählt. So kann z.B. die Anzahl der veränderten oder gelesenen Seiten mit Hilfe der Accessed- und Dirty-Flags bestimmt werden.

Im Longmode beginnen alle Durchläufe mit der PML4-Tabelle (siehe Zeile 21). Die Funktion `traverse` durchläuft die aktuelle Tabelle und führt für jeden Eintrag die gewünschte Operation aus. Befindet sie sich noch nicht in der niedrigsten Tabelle (PGT) ruft sie sich für alle existierenden Einträge, die keine Hugepage darstellen wieder rekursiv auf. Dazu benötigt sie die Adresse der Kind-Tabelle wird über die Funktion `get_child_entry()` bestimmt. Wurden alle Einträge der aktuellen Tabelle durchlaufen kehrt `traverse` wieder zur übergeordneten Tabelle zurück. `traverse()` ist dabei als verschachtelte (engl. nested) Funktion innerhalb des Gültigkeitsbereich von `page_stats()` deklariert. Damit hat sie Zugriff auf den aktuellen Zählerstand `int stats[12]` der Flags.

```
1  void page_stats() {
2    int stats[12] = { 0 };
3
4    void traverse(int level, page_entry_t* entry) {
5      page_entry_t* stop = entry + PAGE_MAP_ENTRIES;
6      for (; entry != stop; entry++) {
7        if (*entry & PG_PRESENT) {
8          if (level && !(*entry & PG_PSE))
9            traverse(level-1, get_child_entry(entry));
10         else {
11           // increment stat counters
12           for (int i=0; i<12; i++) {
13             if (*entry & (1 << i))
14               stats[i]++;
15           }
16         }
17       }
18     }
```

```
19    }
20
21    traverse(PAGE_MAP_LEVELS-1, current_map);
22
23    // print results
24    for (int i=0; i<12; i++)
25      kprintf("%s:␣%lu\n", flag[i], stats[i]);
26  }
```

Listing 2.7: Beispiel eines einfachen Durchlaufs durch die Seitentabllen.

2.4.6. Kopieren und Löschen von Seitentabellen

Der gerade beschriebene Durchlauf wurde aufgrund seiner Einfachheit als Beispiel gewählt. Zwei Weitere (`copy_page_map()` und `map_region()`) befinden sich im Anhang A.

Für das Kopieren der Seitentabellen in einen neuen Prozess ist ein kleiner Trick nötig. Durch die Selbstreferenzierung sind immer nur die Seitentabellen des aktuellen Prozesses sichtbar. Möchte man diese kopieren müssen jedoch gleichzeitig auch die Tabellen des neuen Prozesses eingeblendet werden. MetalSVM erstellt dazu neben der Selbstreferenzierung für den aktuellen Prozess noch zwei weitere: Die Einträge 510 und 509 der aktuellen PML4-Tabelle zeigen auf die PML4-Tabellen des Quell- bzw Zielprozesses. Der nun folgende Durchlauf startet dann parallel auf diesen Einträgen (siehe Quellcode A.2). Mit diesem Ansatz können die Seitentabellen jedes beliebigen Prozesses kopiert werden. Es muss sich dabei nicht um den aktuellen Prozess handeln.

Für das Löschen der Seitentabellen gilt ähnliches. Werden die Seitentabellen des laufenden Prozesses entfernt und durch das BS überschrieben, löst dies einen sogenannten Triple-Fault aus: Der Prozessor versucht auf eine virtuelle Adresse zuzugreifen. Aufgrund der fehlenden Seitentabellen löst das einen Page-Fault aus. Auch der Page-Fault-Handler kann nicht mehr ausgeführt werden, da die Tabellen des Kernel Code nicht mehr vorhanden sind. Aus diesem Zustand kann sich das System nicht mehr retten. Der einzige Ausweg ist ein Neustart.

2.4.7. Beispiel eines Speicherabbildes

Quick Emulator (QEMU) besitzt mit dem Befehl **`info mem`** die Fähigkeit den aktuellen Zustand der Seitentabellen in einem Page-Dump auszugeben. Auch für MetalSVM wurde ähnliches mit der Funktion `page_dump()` implementiert. Tabelle 2.6 zeigt einen Speicherauszug nach dem Booten des Kernels in QEMU (siehe

Start	Ende (Hex)	Seiten	Flags	Verwendung
0x9000	0xA000	1	-g-w	Multiboot
0xB8000	0xBB000	1	-g-w	VGA, LAPIC + IOAPIC
0xC0000	0xD2000	18	-g-w	Kernel Heap
0xF1000	0xF2000	1	-g-w	MP Table
0x100000	0x8C3000	1987	xg-w	Kernel
0x8C3000	0xA34000	369	-g-w	InitRD + Heap
0x40200000	0x40211000	17	x-uw	User Space Code
0x40290000	0x40291000	1	--uw	User Space Data
0x80400000	0x80408000	8	--uw	User Space Stack
0xFFFFFF8000000000	0xFFFFFF8000200000	512	xg-w	PGTs für Kernel Space
0xFFFFFF8000201000	0xFFFFFF8000202000	1	x-uw	PGT für User Space Code + Data
0xFFFFFF8000402000	0xFFFFFF8000403000	1	--uw	PGT für User Space Stack
0xFFFFFFFFC0000000	0xFFFFFFFFC0001000	1	xg-w	PGD für Kernel Space
0xFFFFFFFFC0001000	0xFFFFFFFFC0002000	1	x-uw	PGD für User Space Code + Data
0xFFFFFFFFC0002000	0xFFFFFFFFC0003000	1	--uw	PGD für User Space Stack
0xFFFFFFFFFFE00000	0xFFFFFFFFFFE01000	1	x-uw	PDPT
0xFFFFFFFFFFFFF000	0x10000000000000000	1	---w	PML4

Tabelle 2.6.: Page-Dump nach einer User-Space-Anwendung von MetalSVM.

Abschnitt 5.1). Der untere Teil der Tabelle zeigt die Seitentabellen, die durch die Selbstreferenzierung automatisch im Page-Dump erscheinen. Außerdem zeigt sie die 512 Seitentabellen des Kernels, die statisch im Kernel angelegt sind.

Abbildung D.2 im Anhang visualisiert diesen virtuellen Adressraum in Form eines Speicherabbildes. Dabei bezieht sich die Abbildung auf die 64-Bit-Version des Kernels. Die angegebenen Adressen der Advanced Programmable Interrupt Controllers (APICs), Multiboot-Strukturen oder der Multi-Processing-Tabellen sind dabei nur für das in Kapitel 5.2 vorgestellte Testsystem gültig.

2.5. Vergleich weiterer Architekturen

Neben den gerade vorgestellten Seitentabellen gibt es noch andere Ansätze Paging umzusetzen. Häufig werden auch Kombinationen oder Mischformen der hier vorgestellten Techniken benutzt (Intel Itanium). Zwei dieser Alternativen werden im Folgenden kurz vorgestellt. Tabelle 2.7 gibt einen Überblick über die Paging relevanten Kenndaten gängiger Architekturen.

2.5.1. Inverse Seitentabelle

Abbildung 2.15 zeigt das Funktionsprinzip von inversen Seitentabellen. Im Vergleich zu normalen Seitentabellen, die für jede virtuelle Seite einen Eintrag besitzen, speichert die inverse Seitentabelle lediglich einen Eintrag pro physikalischer Kachel. Dies hat den Vorteil, dass die Größe der benötigten Tabelle nur von der Größe des ASP abhängig ist. Prozesse werden so nicht mehr durch getrennte Seitentabellen, sondern über die PID-Spalte der Inverted Page Table (IPT), voneinander isoliert. Auf diese Weise benötigen gerade große Serversysteme, die viele Prozesse am Laufen haben, weniger Speicherplatz für die Tabellen. Beispiele für den Einsatz dieser Technik sind die PowerPC und Itanium Architekturen.

2.5.2. Software TLB

Abbildung 2.16 zeigt die Arbeitsweise eines durch Software verwalteten Translation Lookaside Buffer. Diese Systeme besitzen keine Hardware für das Durchsuchen von Seitentabellen (Pagetable-Walk). Jeder TLB-Miss löst einen Seitenfehler aus, der die Adressübersetzung in Software realisieren muss. Das Ergebnis dieser Übersetzung wird anschließend für folgende Zugriffe auf die gleiche Adresse im TLB zwischengespeichert. Mit dieser Methode ist es dem BS freigestellt wie es die Zuordnung von virtuellem zu physikalischen Adressraum realisiert. Ein Beispiel für den Einsatz dieser Technik ist die MIPS-Architektur.

[1] Falls Page Size Extension (PSE) aktiviert sind.
[2] Abhängig von der Implementierung.
[3] Mögliche Erweiterung.

Architektur / Prozessor	ISA	Level	Seitengrößen	#Bits virtuell	#Bits physikalisch
Intel x86 [11]	CISC-32	1, 2	4 KiB, 4 MiB	10+10+12 = 32	32 (32+4[1])
Intel x86-64 [11]	CISC-64	2, 3 oder 4	4 KiB, 2 MiB, (1 GiB)	9+9+9+9+12 = 48	≤ 52[2]
ARMv7-A [1]	RISC-32	bis zu 3	4, 64 KiB, 1, 16 MiB	32	≤ 40[2]
ARMv8-A [2]	RISC-64	bis zu 4	4, 64 KiB	9+9+9+9+12 = 48	≤ 48[2]
SuperSPARC II [20], [22]	RISC-32	3	4, 256 KiB, 16 MiB, 4 GiB	8+5+5+12 = 32	36
UltraSPARC T1 [21], [23]	RISC-64	Software TLB	8, 64 KiB, 4, 256 MiB	48	40
SPARC64 V [21], [7]	RISC-64	Software TLB	8, 64, 512 KiB, 4 MiB	64	43
MIPS32 [15]	RISC-32	Software TLB	1 KiB - 256 MiB		36
MIPS64 [16]	RISC-64	Software TLB	1 KiB - 256 MiB		59
PowerPC [6]	RISC-32	hashed IPT	4 KiB		≤ 32
PowerPC [12]	RISC-64	hashed IPT	4 KiB		62
Intel Itanium	CISC-64	hashed IPT	4, 8, 16, 64, 256 KiB,	64	≤ 50 (≤ 63[3])
			1, 4, 16, 64, 256 MiB, 4 GiB		≤ 62
Alpha [3]	RISC-64	3	8 KiB	10+10+10+13 = 43	43
			16 KiB	11+11+11+14 = 47	45
			32 KiB	12+12+12+15 = 51	47
			64 KiB	13+13+13+16 = 55	48
VAX-11 [5]	CISC-32	2	512 B	2+21+9	30

Tabelle 2.7.: Vergleich des Pagings gängiger Architekturen.

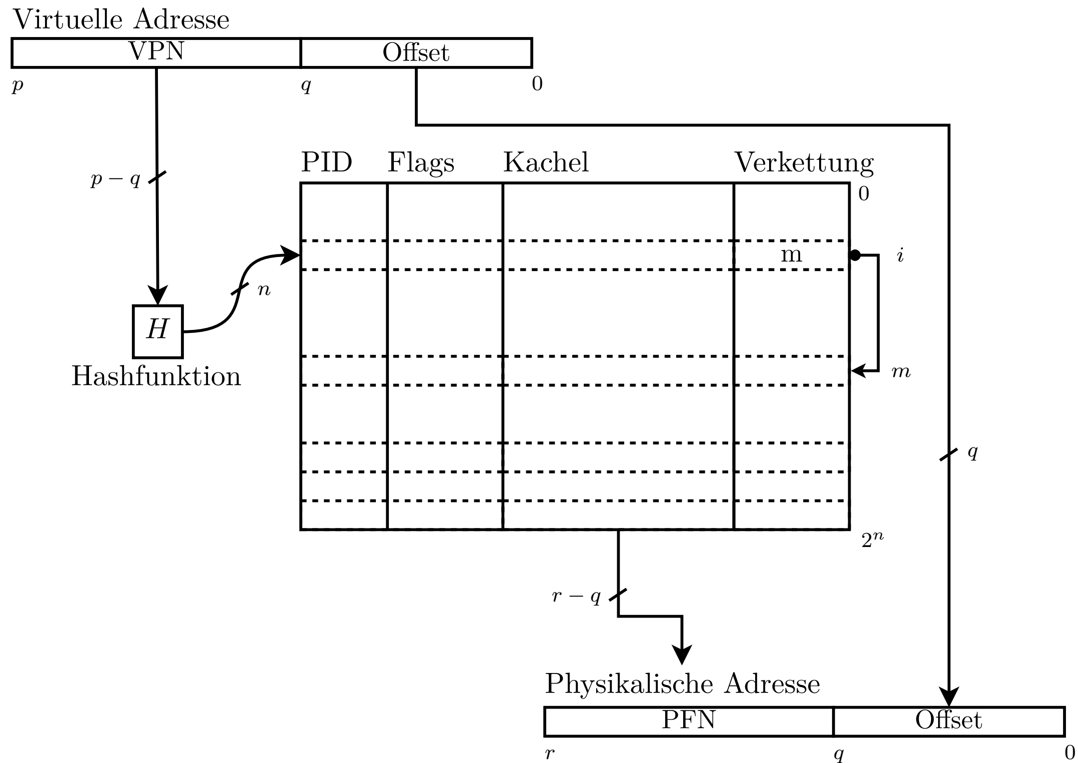


Abbildung 2.15.: Schema einer inversen Seitentabelle mit Hashfunktion H.

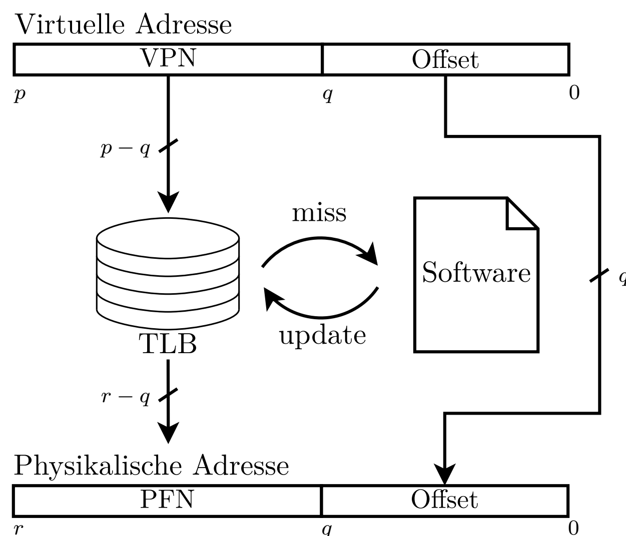


Abbildung 2.16.: Adressübersetzung mit Software-verwaltetem TLB.

3. Virtual Memory Areas

Für eine flexible und portable Speicherverwaltung ist neben Paging eine weitere Abstraktionsebene nötig. Wie im vorherigen Kapitel beschrieben ist Paging stark an die zugrunde liegende Architektur bzw. Memory Management Unit (MMU) geknüpft. Auch wenn die meisten Architekturen Paging ähnlich umsetzen, unterscheiden sie sich in einigen Parametern:

- Größe der Seiten
- Anzahl der Tabelleneinträge
- Anzahl der Ebenen
- Format der Tabelleneinträge
- Verfügbare Eigenschaften (Flags)

3.1. Grundlagen

Die in diesem Kapitel vorgestellten Virtual Memory Areas (VMAs) versuchen diese architekturspezifischen Eigenschaften auszugleichen. Virtual Memory Areas fassen kontinuierliche Speicherbereiche zu logischen Einheiten zusammen, die mit weiteren Flags und Eigenschaften versehen werden. Während Paging die Zuordnung zwischen virtuellem- und physikalischem Adressraum festlegt, repräsentieren die VMAs nur den Zustand des virtuellen Adressraums. Beispiele hierfür sind die Stacks, der Heap, der Kernel, die Init Ramdisk (InitRd) oder Sektionen einer User-Space-Anwendung (`.code`, `.data`, `.bss`). Diese Bereiche dürfen jede beliebige Größe und Position innerhalb des VA annehmen, sich dabei jedoch nicht überlappen. Die zuvor durch das Paging festgelegten Einschränkungen der Seitengröße und Ausrichtung im Speicher treffen auf VMAs nicht mehr zu.

3.1.1. Bezug zum PA / Paging

Prinzipiell lässt sich das Konzept der VMAs auch auf den PA übertragen. Beide Adressräume erfordern eine konsequente Protokollierung belegter und freier Speicherbereiche. Auch die Kacheln des physikalischen Adressraums könnten mit

weiteren Eigenschaften belegt werden. So mag es hilfreich mit einem Referenzzähler die Anzahl der Nutzungen einer Seite zu zählen. Weiterhin könnte man Bereiche des PA bestimmten Zonen eines NUMA-Systems (Non-Uniform Memory Access) zuordnen.

MetalSVM nutzt zur Verwaltung des physikalischen Adressraums (PA) eine Bitmap. Dieser recht einfache Ansatz kodiert belegte Kacheln mit einem gesetzten – freie Kacheln mit einem nicht gesetzten – Bit. Weitere Eigenschaften können daher nicht gespeichert werden. Für die Verwaltung des VA ist eine Bitmap jedoch nicht praktikabel. So würde die x86-64-Architektur, welche einen VA von 2^{48} Bit besitzt, mit Gleichung 3.1 eine Bitmap der Größe 8 GiB pro Prozess benötigen.

$$MAX_TASKS \cdot \frac{2^{VIRT_BITS}}{8 \cdot PAGE_SIZE} \tag{3.1}$$

Linux nutzt ein statisches Array (`mem_map`) zur Verwaltung des PA [8]. Jede Kachel wird durch ein Element (`struct page`) in diesem Array beschrieben. Die Elemente enthalten neben einem Referenzzähler auch weitere Eigenschaften zu der ihr zugehörigen Kachel:

> Each physical page in the system has a struct page associated with it to keep track of whatever it is we are using the page for at the moment.[1]
> [...]

3.2. Implementierung

Es bietet sich an, die belegten Speicherbereiche (VMA) in einer dynamischen Datenstruktur abzulegen. Dafür bieten sich bspw. balancierte Bäume (AVL oder R/S) oder verkettete Listen an. Statische Datenstrukturen wie z.B. ein Array sind ungeeignet, da das Erstellen einer neuen VMA das Verschieben der restlichen Einträge erfordern würde. Eine dynamische Struktur ermöglicht dahingehend flexibler. Die Anzahl der Regionen muss im Vorhinein nicht bekannt sein. Die Regionen können leicht in der Liste verschoben, entfernt und eingefügt werden.

MetalSVM nutzt der Einfachheit halber eine doppelt-verkettete Liste zum Verwalten der VMAs. Quellcode 3.2.2 zeigt die Deklaration eines Listenelementes. Dieses repräsentiert eine VMA und besteht aus zwei Zeigern auf benachbarte Listenelemente, Start- und Endadresse der Region sowie weiteren Flags und Eigenschaften.

[1] `linux/include/linux/mm_types.h`

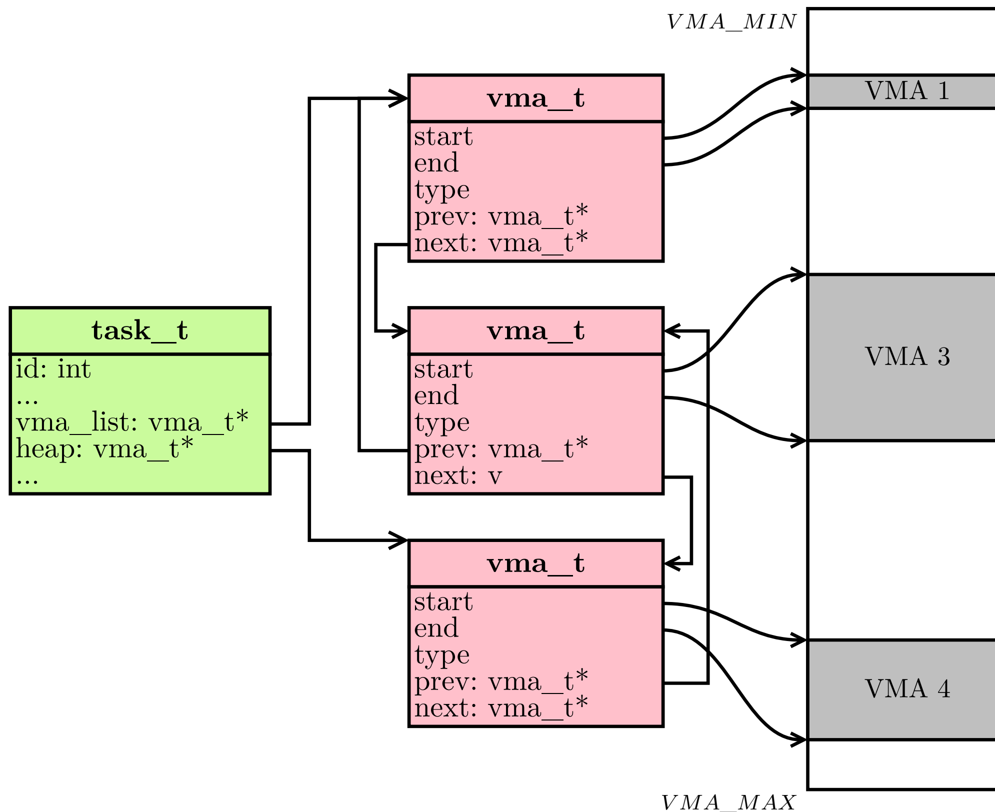


Abbildung 3.1.: VMA-Liste eines Prozesses im User-Space.

3.2.1. Henne-Ei-Problem

Die Verwendung einer dynamischen Datenstruktur birgt jedoch einen Nachteil. Für das Anlegen einer neuen VMA muss für das Listenelement zunächst dynamischer Speicher angefordert werden. Diese Speicheranforderungen werden aus dem Heap bedient (siehe Kapitel 4). Der Heap des Kernels ist jedoch selbst wiederum eine VMA, die zuerst erzeugt werden muss. Diese umgangssprachlich als „Henne-Ei-Problem“ bekannte Problematik lässt sich umgehen, indem die VMA des Heaps statisch im Kernel abgelegt wird. Diese VMA wird mit einer Größe von 0 B initialisiert und bei folgenden Speicherangeforderungen schrittweise vergrößert. Das Kapitel 4 stellt die Methoden zur Verwaltung des Heaps ausführlicher vor.

3.2.2. MetalSVM

Neben dem Heap gibt es noch eine Reihe weiterer VMAs, die beim Starten angelegt werden. Tabelle 3.1 zeigt diese für das in Kapitel 5.2 vorgestellte Testsystem. Wie in der Tabelle zu erkennen ist, gibt es getrennte Listen für den Kernel- und User-

Liste	Region	Start	Größe
Kernel-Space	SMP Boot Code	0x7000	4 KiB
	Multiboot Information	0x9000	4 KiB
	Multiprocessing Table	0xF1000	4 KiB
	VGA Video Speicher	0xB8000	4 KiB
	LAPIC	0xB9000	4 KiB
	IOAPIC	0xBA000	4 KiB
	Heap (Buddysystem)	0xC0000	*wachsend* ↓
	Kernel	0x100000	*variabel*
	Init Ramdisk	0x8c3000	*variabel*
User-Space	.text, .data, .bss	0x40200000	*variabel*
	Heap	0x40203523	*wachsend* ↓
	Stack	0x80400000	*wachsend* ↑

Tabelle 3.1.: Die VMA-Regionen von MetalSVM.

Space. Jeder Prozess besitzt seine eigene VMA-Liste. Diese ist Bestandteil des Prozesskontrollblocks (`task->vma_list`). Für den Kernel ist eine einzige VMA-Liste ausreichend (`vma_list`).

Es folgt eine kurze Beschreibung der Funktionen und Datentypen[2].

`vma_t` ist der Datentyp eines Listenelements, das eine VMA repräsentiert:

```
typedef struct vma {
  size_t start;
  size_t end;
  uint32_t flags;
  struct vma* next;
  struct vma* prev;
} vma_t;
```

`vma_init()` wird beim Booten einmalig aufgerufen. Sie initialisiert die VMA-Liste des Kernels mit den Einträgen aus Tabelle 3.1. Diese Einträge können sich von System zu System unterscheiden und werden daher erst zur Laufzeit initialisiert.

`vma_alloc(size, flags)` sucht nach einem freien Speicherbereich der Größe `size` im VA. Hierbei wird die Liste der VMAs linear durchlaufen und der ers-

[2] `metalsvm/mm/vma.c` und `metalsvm/include/metalsvm/vma.h`.

te freie und passende Bereich gewählt (engl. first fit). Mit dem Flag `VMA_USER` wird der Bereich in die VMA-Liste des aktuellen Prozesses eingefügt.

`vma_add(start, end, flags)` fügt eine neue VMA beginnend bei Adresse `start` bis `end` in die VMA-Liste ein. Ähnlich wie bei der Funktion `vma_alloc()` kann für das Flag `VMA_USER` die Liste gewählt werden. Zudem können mit `flags` weitere Eigenschaften kodiert werden. Diese Eigenschaften sind zur Zeit nur informeller Natur und werden vom Betriebssystem (BS) nicht weiter beachtet.

`vma_free(start, end)` entfernt eine VMA aus der Liste und gibt den dynamischen Speicher des Listenelementes frei.

`vma_dump()` gibt eine Liste aller VMAs aus. Dazu werden sowohl die Liste des Kernels als auch die VMA-Liste des aktuellen Prozesses durchlaufen.

`copy_vma_list(src, dest)` kopiert die VMA-Liste des Prozesses `src` zum Prozess `dest`. Diese Funktion wird für das Duplizieren (engl. fork) von Prozessen benötigt. Ein Fork des Prozesses kopiert sowohl die Seitentabellen als auch den Inhalt der Seiten sowie die VMA-Liste.

`drop_vma_list(task)` gibt alle dynamisch angeforderten Listenelemente der VMA Liste von Prozess `task` frei. Diese Funktion wird beim Beenden eines Prozesses ausgeführt. Auch der `execve()` Systemaufruf benutzt sie um eine exakte Kopie aller Regionen anzulegen.

Zwei weitere Funktionen kombinieren die Funktionen des Pagings mit der Verwaltung des VA sowie PA:

`palloc(size, flags)` fordert Kacheln aus dem PA als auch Seiten aus dem VA an. Zu diesem Zweck wird eine neue VMA erzeugt. Diese umfasst zwangsläufig immer das Vielfache der Seitengröße. Anschließend werden die Kacheln mit `map_region()` in die erzeugte VMA eingeblendet. Für kleinere Speicherbereiche sollte die Funktion `kmalloc()` verwendet werden, die Speicher aus dem Heap allokiert[3].
Siehe: `get_pages()` → `vma_alloc()` → `map_region()`.

`pfree(addr, size)` gibt zuvor mit `palloc()` angeforderte Bereiche wieder frei.
Siehe: `vma_free()` → `put_pages()` → `unmap_region()`.

[3]Genau genommen wird der Heap über `palloc()` erzeugt.

4. Heap Speicher

Der Heap fasst dynamisch verwalteten Speicher zusammen. Dieser existiert sowohl im Kernel-Space als auch für jeden Prozess im User-Space. Die C-Standard-Bibliothek (libc)[1] bietet mit den Funktionen `malloc(size)` und `free(ptr)` zwei Schnittstellen zum Anfordern und Freigeben von dynamischem Speicher im User-Space. Für C++ wurden diese Funktionen mit den Operatoren `new` und `delete` Bestandteil des Sprachstandards.

Die libc nutzt in ihrer Implementierung einen Systemaufruf namens `sbrk(incr)`. Dieser vergrößert bzw. verkleinert die Größe des User-Space-Heaps um `incr` Byte. Aus Sicht des Kernels ist der Heap von User-Space-Prozessen ein kontinuierlicher Speicherbereich. Dieser wird vom Kernel durch eine Virtual Memory Area (VMA) repräsentiert. Nur die Standardbibliothek hat ein genaueres Bild der Speicheraufteilung.

Ohne diesen Systemaufruf (engl. syscall), der im Kernel-Space nicht zur Verfügung steht, kann die libc keinen dynamischen Speicher verwalten. Da es jedoch vorteilhaft wäre, auch im Kernel dynamischen Speicher nutzen zu können, stellt das folgende Kapitel einen einfachen Algorithmus vor, der diese Aufgabe übernimmt.

4.1. Grundlagen

Sowohl die libc als auch der Kernel müssen den ihnen zu Verfügung stehenden Heap verwalten. Dabei müssen sie Speicheranforderungen (Allokationen) variabler Größen erlauben, ohne bei häufigem Anfordern und Freigeben von Blöcken den Speicher zu fragmentieren (siehe Abschnitt 2.1.2).

Eine weitere Anforderung ist eine effiziente Allokation und Freigabe von Blöcken. Diese sollte möglichst in konstanter Komplexität $\mathcal{O}(1)$ erfolgen. Es gilt also immer einen gewissen Vorrat an freien Speicherbereichen vorzuhalten, sodass Anfragen auf diesen zurückgreifen können. Das Vergrößern des Heaps löst in der Regel einen Systemaufruf aus oder erfordert das Anlegen neuer Seitentabellen. Diese unvorhersehbaren Einbußen in der Performance möchte man in der Regel vermeiden.

Linux nutzt für diesen Zweck den sogenannten SLAB-Allocator [8]. Dieser fasst Datenstrukturen gleichen Typs in sogenannten „Pools“ zusammen. Da alle Ele-

[1] Genauer genommen ihre Implementierungen: die von MetalSVM genutzte newlib bzw. ulibc, glibc etc.

mente eines Pools die gleiche Größe besitzen, können diese somit sehr kompakt gespeichert werden. Beispiele für diese Datenstrukturen sind iNodes, Prozesskontrollblöcke, Sockets und VMAs (siehe `/proc/slabinfo`). Weiterhin versucht der SLAB-Allokator die Vorteile der CPU-Caches auszunutzen, indem er eine Reihe von Objekten bereits vorab allokiert.

4.1.1. Buddysystem

Das von Kenneth C. Knowlton vorgestellte Buddysystem teilt den Heapspeicher in Blöcke der Größe 2^x Byte [13]. Grafisch lässt sich dieses Buddysystem als binärer Baum oder mit Hilfe der DIN-Papierformate darstellen (siehe Abb. 4.1). Im Folgenden werden die Blöcke des Buddysystems als Buddies bezeichnet.

Jede Anforderung (engl. allocation) wird mit dem nächst größeren Buddy der Größe 2^x Byte bedient. Ist kein Buddy dieser Größe mehr vorhanden, werden größere Buddies sukzessive geteilt bis die gewünschte Größe erreicht wurde. Beim Freigeben wird umgekehrt vorgegangen. Freie, jeweils benachbarte, Buddies werden wieder zu einem nächst größeren Buddy zusammengefasst (engl. merged). Durch das Zusammenfassen wird eine externe Fragmentierung weitestgehend vermieden. Neben der einfachen Implementierung ist dies einer der größten Vorteile des Buddysystems.

Als Nachteil ist die interne Fragmentierung zu nennen, die durch das Buddysystem entsteht. Die festen Blockgrößen führen zwangsläufig dazu, dass der Speicher eines Buddies nicht vollständig genutzt werden kann. Es zeigt sich, dass es durchaus sinnvoll ist, einen gewissen Grad an interner (zugunsten geringerer externer) Fragmentierung zu tolerieren. Die interne Fragmentierung ist immer auf maximal eine halbe Blockgröße beschränkt. Geht man von gleichverteilten Blockgrößen der Anfragen aus, liegt die interne Fragmentierung im Schnitt bei einem Viertel der Blockgröße.

Das von Knuth vorgestellte Buddysystem teilt Blöcke immer im Verhältnis einzu-eins. Neben diesem *binären* Ansatz gibt es auch Alternativen, die eine Teilung gemäß der Fibonacci-Zahlen oder dem goldenen Schnitt vorsehen. Auch eine Teilung in drei oder mehr Buddies wird eingesetzt [19]. Diese *gewichteten* oder auch *tertiären* Systeme haben den Vorteil, dass sie eine größere Auswahl an Blockgrößen zur Verfügung stellen. Außerdem minimieren sie in der Folge die interne Fragmentierung, jedoch auf Kosten der Komplexität.

Auch der Linux-Kernel nutzt auch ein binäres Buddysystem [8]. Jedoch findet es dort seinen Einsatz zur Verwaltung physikalischer Kacheln und nicht des Heaps (siehe `/proc/buddyinfo`). Für dynamische Datenstrukturen im Kernel wird der oben vorgestellte SLAB-Allokator eingesetzt. Auf diese Weise wird auch die interne Fragmentierung des Buddysystems weiter reduziert.

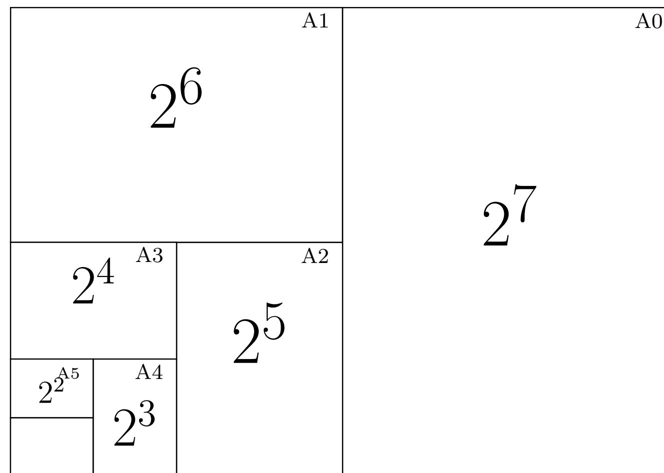


Abbildung 4.1.: Vergleich des binären Buddysystems mit den DIN Papierformaten.

4.2. Implementierung

Für eine effiziente Allokation werden freie Buddies in Listen abgelegt. Für jede Größe (Zweierexponent) der Buddies existiert eine eigene Liste. Beim Anfordern von Speicher wird ein Buddy aus der entsprechenden Liste entfernt. Eine Freigabe fügt den Buddy wieder zurück in die Liste ein.

MetalSVM nutzt dazu einfach verkettete Listen (`buddy_lists[]`, siehe Abb. 4.2). Da die Buddies selbst freien Speicher repräsentieren, kann dieser für die Elemente der verketteten Liste genutzt werden. Jeder Buddy ist mit einem Präfix versehen (siehe Deklaration 4.2). Die Präfixe sind den freien Speicherbereichen immer vorangestellt. Für freie Buddies enthält dieses Präfix einen Zeiger auf das nächste Listenelement. Das Präfix eines belegten Buddy enthält dessen Größe und einen Marker. Diese sind nötig um den Buddy bei der Freigabe wieder in die korrekte Liste einsortieren zu können. Zudem muss der Marker eines gültigen Buddies (engl. magic) den Wert `BUDDY_MAGIC` (`0xBABE`) enthalten. Die Kontrolle des Präfixes verhindert das Freigeben von Speicherbereichen, die nicht mit `kmalloc()` angefordert wurden. Gleichzeitig stellt es sicher, dass die im Präfix gespeicherte Größe des Buddies noch gültig ist. Würde das Präfix durch einen Programmierfehler überschrieben werden, wäre eine korrekte Freigabe des Bereichs nicht mehr möglich.

Es folgt eine kurze Beschreibung der Funktionen, Deklarationen und Konstanten des Heap-Allokators:

`BUDDY_MAX` definiert die maximale Größe eines Buddies als Zweierexponent. Für MetalSVM liegt dieser Wert bei 32, womit die Größe eines Speicherblocks auf $2^{32} -$ `sizeof(buddy_t)` ≈ 4 GiB begrenzt ist. Größere Speicheranforderungen können durch den Heap-Allokator nicht bedient werden.

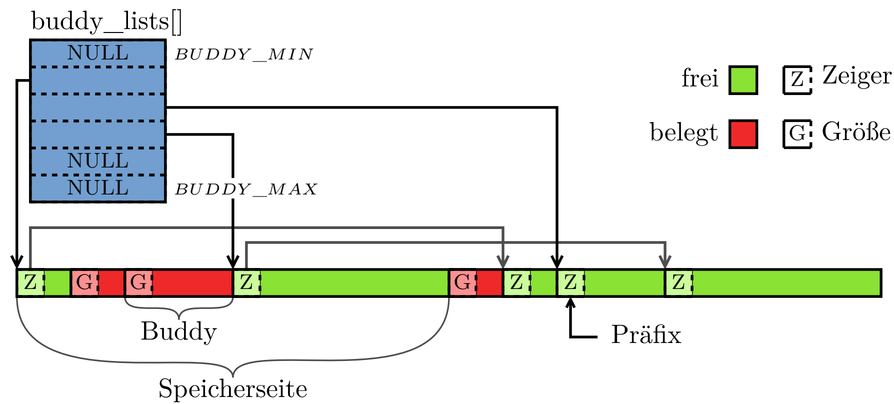


Abbildung 4.2.: Die verkettete Liste des Buddysystems.

`BUDDY_MIN` definiert die minimale Größe eines Buddies als Zweierexponent. Für MetalSVM liegt dieser Wert bei 3, womit die Größe eines Buddies mindestens $2^3 = 8$ Byte beträgt. Da jeder Buddy implizit auch ein Präfix enthält müssen die Buddies mindestens größer als das Präfix sein.

`BUDDY_ALLOC` definiert die Größe eines neuen Buddies, der per `palloc()` angefordert wird. Sollten die Buddylisten keine ausreichend großen Buddies mehr enthalten, werden neue Speicherseiten für den Heap angefordert. Diese neuen Speicherseiten werden dann als neuer Buddy in die Listen eingefügt, der anschließend wieder geteilt wird. Für MetalSVM liegt dieser Wert bei 16, sodass immer $2^{16} = 16 \cdot 2^{12} = 64\,\text{KiB}$ angefordert werden.

`BUDDY_MAGIC` ist definiert als `0xBABE` und markiert einen gültigen Buddy.

`buddy_t` ist das Präfix welches jedem Buddy vorangestellt wird.

```
1 typedef union buddy {
2   union buddy* next;
3   struct {
4     uint8_t exponent;
5     uint16_t magic; /* should be equal to BUDDY_MAGIC */
6   } prefix;
7 } buddy_t;
```

`buddy_lists` ist ein Array von verketteten Buddylisten. Die Anzahl der Listen wird durch `BUDDY_MAX - BUDDY_MIN + 1` vorgegeben.

```
#define BUDDY_LISTS (BUDDY_MAX - BUDDY_MIN + 1)
static buddy_t* buddy_lists[BUDDY_LISTS] = { NULL };
```

buddy_get(exp) liefert einen freien Buddy der Größe 2^{exp} Byte. Diese Funktion wird dabei rekursiv aufgerufen, um größere Buddies sukzessive aufzuteilen. Sollten keine ausreichend großen Buddies mehr vorhanden sein, werden mit `palloc()` neue Seiten für den Heap angefordert.

```
buddy_t* buddy_get(int exp) {
  buddy_t** list = &buddy_lists[exp-BUDDY_MIN];
  buddy_t* buddy = *list;
  buddy_t* split;

  if (buddy)
    *list = buddy->next;
  else if (exp >= BUDDY_ALLOC &&
     ↪ !buddy_large_avail(exp))
    buddy = (buddy_t*) palloc(1<<exp, 0);
  else {
    buddy = buddy_get(exp+1);
    if (buddy) {
      split = (buddy_t*) ((size_t) buddy + (1<<exp));
      split->next = *list;
      *list = split;
    }
  }

  return buddy;
}
```

buddy_put(buddy) fügt einen Buddy zurück in die Liste freier Buddies ein. In dieser Version werden jedoch noch keine benachbarten Buddies zusammengefasst.

```
void buddy_put(buddy_t* buddy) {
  buddy_t** list =
     ↪ &buddy_lists[buddy->prefix.exponent-BUDDY_MIN];
  buddy->next = *list;
  *list = buddy;
}
```

kmalloc(size) fordert einen dynamischen Speicherblock der Größe `size` Byte aus dem Kernel-Heap an. In der letzten Zeile wird die Adresse des Speicherblocks zurückgegeben. Die Addition „versteckt“ hier das Präfix vor dem Programmierer.

```
void* kmalloc(size_t sz) {
  // add space for the prefix
  sz += sizeof(buddy_t);

  int exp = buddy_exp(sz);
  if (BUILTIN_EXPECT(!exp, 0))
    return NULL; // invalid size requested

  buddy_t* buddy = buddy_get(exp);
  if (BUILTIN_EXPECT(!buddy, 0))
    return NULL; // out of memory

  // setup buddy prefix
  buddy->prefix.magic = BUDDY_MAGIC;
  buddy->prefix.exponent = exp;

  // pointer arithmetic: we hide the prefix
  return buddy+1;
}
```

kfree(addr) gibt den zuvor mit `kmalloc()` angeforderten Speicher wieder frei.

```
void kfree(void *addr) {
  buddy_t* buddy = (buddy_t*) addr - 1; // get prefix

  // check magic
  if (BUILTIN_EXPECT(buddy->prefix.magic !=
     ↪ BUDDY_MAGIC, 0))
    return;

  buddy_put(buddy);
}
```

5. Toolchain und Hardware

Dieses Kapitel stellt die verwendete Software, Werkzeuge (engl. toolchain) und die zum Testen verwendete Hardware vor.

5.1. Quick Emulator

Für die Arbeit am MetalSVM-Kernel wurde vornehmlich der Quick Emulator (QEMU) eingesetzt. Dieser beherrscht neben der Emulierung der x86-Architektur von Intel/AMD auch weitere Systeme wie bspw. PowerPC, ARM, Alpha, CRIS, LatticeMico32, m68k (Coldfire), MicroBlaze, MIPS, Moxie, SH-4, S/390, Sparc32/64, OpenRISC, Unicore und Xtensa[1]. Mit der Verfügbarkeit spezieller Hardwareunterstützung wurde QEMU um Virtualisierungsfunktionen erweitert. Zwei Vertreter dieser Virtualisierungsfunktionen sind Xen und Kernel-based Virtual Machine (KVM).

Während der Entwicklung des Kernels wurde eine reine Software-Emulation der x86-64 Architektur genutzt. Im Gegensatz zur hardwaregestützten Virtualisierung ermöglicht diese einen tieferen Einblick in den Zustand des Systems. So kann bspw. der physikalische Adressraum untersucht werden oder der aktuelle Zustand der Seitentabellen und des Translation Lookaside Buffer (TLB) dargestellt werden. Für verwertbare Messergebnisse musste jedoch auf reale Hardware zurückgegriffen werden.

5.1.1. Monitoring Konsole

QEMU besitzt mit der „Monitoring Console“ ein hilfreiches Werkzeug zur Steuerung, Fehlersuche und Überwachung des emulierten Systems. Die folgenden Befehle waren für die Arbeit an der virtuellen Speicherverwaltung von besonderer Hilfe:

`info mem` Zeigt die durch Paging eingeblendeten Seiten des aktuellen Prozesses. Dieser Befehl ist vergleichbar mit der `page_dump()` Funktion im MetalSVM-Kernel.

[1] `http://git.qemu.org/`.

info tlb Zeigt den aktuellen Zustand des Translation Lookaside Buffer. Neben der physikalischen Adresse werden hier auch die effektiven Flags der Einträge angezeigt.

info registers Zeigt die aktuellen Werte aller Register der CPU. Dieser Befehl hat gegenüber dem GDB den Vorteil, dass wirklich alle Register ausgegeben werden. Über den GNU Debugger (GDB) kann z.B. nicht auf das für das Paging relevante Control Register 3 (CR3) zugegriffen werden.

sum Berechnet eine Prüfsumme über einen Ausschnitt des virtuellen Adressraums (BSD-kompatibel).

[**p**]**memsave** Extrahiert einen Bereich des virtuellen/physikalischen Speichers in eine Datei auf den Hostsystem.

x[**p**] Gibt Ausschnitte des virtuellen/physikalischen Speichers auf der Konsole aus. Die Syntax dieses Befehls ist an das Format des GDB angelehnt.

gdbserver Startet eine entfernte Debugging-Session. In Verbindung mit dem GDB kann diese zur tiefergehenden Fehlersuche genutzt werden.

5.1.2. Universal Asynchronous Receiver Transmitter

Für die Ein-/Ausgabe wurde eine serielle UART-Schnittstelle genutzt. Diese kann über QEMU emuliert werden und ist ebenfalls im realen Testsystem vorhanden. Um diese nutzen zu können musste ein neuer Treiber im Kernel implementiert[2] werden. Die bisher genutzte Ausgabe über die VGA-Konsole besitzt den Nachteil, dass Messergebnisse nicht protokolliert werden können. Die Ausgaben werden so schnell über den Bildschirm gescrollt, dass hier ein Mensch nicht mehr folgen kann. Die Kommunikation über UART ermöglicht dagegen ein einfaches Speichern und Durchsuchen der Ausgaben (siehe Anhang B).

Der UART-Transceiver wird direkt über die IO-Ports des Prozessors angesprochen. Zum Senden und Empfangen einzelnen Zeichen werden die Instruktionen `out` und `in` genutzt. Tabelle 5.1 enthält die Adressen der IO Ports für die ersten seriellen Schnittstellen eines IBM kompatiblen PCs.

5.1.3. 32/64-Bit-Debugging

Zur Fehlersuche wurde der GNU Debugger (GDB) eingesetzt. Der Debugger kann sich über das `gdbserver`-Protokoll mit entfernten System verbinden. Dieser Mechanismus wird genutzt um das emulierte System von Quick Emulator (QEMU) zu

[2] `metalsvm/arch/x86/kernel/uart.c`.

Port	IO Adresse
COM1	3F8h (Testsystem)
COM2	2F8h (SCC)
COM3	3E8h
COM4	2E8h

Tabelle 5.1.: Die Standard IO-Ports des UART-Transceivers.

debuggen. Nutzt man den integrierten GDB-Stub von QEMU stößt man unweigerlich an die Grenzen des Protokolls. Dieses gestattet keinen Wechsel der Architektur (bzw. Betriebsmodus) während einer Debugging-Sitzung. In der Folge ist ein Wechsel vom 32-Bit-Protected-Mode in den 64-Bit-Longmode nicht möglich. Da gerade dieser Übergang für das Initialisieren der Seitentabellen jedoch von Interesse ist, musste der GDB für diesen Fall angepasst werden. Der dazu nötige Patch findet sich im Anhang (Abschnitt C.1).

5.2. Testsystem

5.2.1. Hardware

Durch die Software-Emulation eignet sich QEMU nicht für Benchmarks. Kontextwechsel und Interrupts auf dem Hostsystem führen zu unvorhersehbaren Verzögerungen im emulierten System. Zudem nutzt QEMU eine sogenannte „SoftMMU“[3]. Diese führt alle Adressübersetzungen durch Software aus und ist daher nicht vergleichbar mit einem hardwarebasierten Pagetable-Walk, der den Translation Lookaside Buffer mit einbezieht. Da der Emulator nicht alle Eigenschaften der Hardware vollständig simulieren kann sind Tests auf echter Hardware nötig. Dazu wurde der Rechner „`aragorn`“ des Instituts genutzt. Tabelle 5.2 fasst die Eigenschaften des Testsystems zusammen.

Dieser Rechner besitzt einen 64-Bit-Prozessor der Core-2-Duo-Mikroarchitektur von Intel. Er besitzt einen virtuellen Adressraum (VA) von 2^{48} Bits und einen physikalischen Adressraum (PA) von bis zu 2^{36} Bits. Die CPU besteht aus zwei Kernen, die kein HyperThreading unterstützen. Die dynamische Taktregelung (Intel Speed Step) sowie Virtualisierungsfunktionen (Intel VT-x) wurden deaktiviert. Um weitere Seiteneffekte auszuschließen wurde das System ohne Festplatten, optische Laufwerke und Grafikkarte betrieben. Für die Ein-/Ausgabe wurde wurde die zuvor beschriebene Kommunikation über die serielle Schnitstelle genutzt. Für die durchgeführten Tests sind unter anderem Details der Caches und TLBs aus

[3] `http://qemu.weilnetz.de/qemu-tech.html#MMU-emulation`.

Mainboard	Asus P5K Green[4]
Arbeitsspeicher	2048 MB DDR2 1333 Mhz
Prozessor	Intel Core 2 Duo E6550 @2,33 GHz
Kerne	2
HyperThreading	nein
Intel Speed Step	deaktiviert
Veröffentlichung	Q3'07
Microarchitektur	Core 2 Duo (Conroe G0 65nm)
Netzwerkkarte	LevelOne (Realtek RT8139-kompatibel)

Tabelle 5.2.: Hardware des Testsystems.

Typ	**Level**	**Größe**	**Assoziativität**
Cache	L1 Instruktionen	$2 \cdot 32$ KiB	8-fach
	L1 Daten	$2 \cdot 32$ KiB	8-fach
	L2 gemeinsam	4 MiB	16-fach
TLB	Instruktionen (2 MiB/4 MiB Seiten)	4/8 Einträge	4-fach
	Instruktionen (4 KiB Seiten)	128 Einträge	4-fach
	Daten (2 MiB/4 MiB Seiten)	16/32 Einträge	4-fach
	Daten (4 KiB Seiten)	256 Einträge	4-fach
	L1 Daten (4 KiB Seiten)	16 Einträge	4-fach
	L1 Daten (2 MiB/4 MiB Seiten)	8/16 Einträge	4-fach

Tabelle 5.3.: TLB und Cachedetails des Testsystems.

Tabelle 5.3 von Interesse[5].

5.2.2. Testablauf

Um das Testen zu vereinfachen wurde der MetalSVM über das Netzwerk gebootet. Dazu wurde eine spezielle Preboot Execution Environment (PXE) Implementierung namens „iPXE“[6] verwendet. Dieser Bootloader kann auf das Boot-ROM gängiger Netzwerkkarten geflasht oder per USB-Stick geladen werden. Er unterstützt das Booten von Linux, Windows und Multiboot-kompatiblen Kerneln. In Ergänzung zum PXE Standard unterstützt iPXE (ehemals Etherboot, gPXE) auch das Laden von Kerneln über HTTP, (T)FTP, iSCSI, FibreChannel, WLAN und InfiniBand. Die eingesetzte Konfiguration ist im Anhang zu finden (siehe C.2). Bei MetalSVM handelt es sich um einen Multiboot-konformen Kernel [17]. Dies erlaubt

[5] `http://ark.intel.com/products/30783/intel-core2-duo-processor-e6550-4m-cache-2_33-ghz-1333-mhz-fsb`.

[6] `http://ipxe.org/`.

das Laden über den Grand Unified Bootloader (GRUB), QEMU oder iPXE.

Um den Arbeitsablauf noch weiter zu optimieren wurde das Zurücksetzen und Starten des Systems automatisiert. Ein dazu entwickeltes Mikrocontroller-Board löst mittels Relais einen Neustart des Testsystems aus. In Verbindung mit der Ein-/Ausgabe über einen RS232-Port und Makefiles konnte so das Testen vollständig automatisiert werden (siehe Anhang C.3).

5.2.3. Performance Monitoring Counter

Für die Evaluierung der Paging-Implementierung sind unter anderem die Anzahl an TLB-Misses, Dauer der Pagetable-Walks oder Cache-Misses von Interesse. Die einzige Möglichkeit diese Informationen zu erhalten sind die Performance Monitoring Counter (PMC). Bei ihnen handelt es sich um einfache Hardware-Zähler, die fortlaufend das durch Auftreten bestimmter Ereignisse in der CPU inkrementiert werden. Das eingesetzte Testsystem besitzt insgesamt fünf dieser PMC. Drei der Zähler sind mit einem festgelegte Ereignisse (engl. fixed function, FF) vorbelegt. Die restlichen beiden können frei wählbaren Ereignissen belegt werden (engl. general purpose, GP). Tabelle 5.4 zeigt die für diese Arbeit relevanten Ereignisse der Core-2-Duo-Mikroarchitektur (`PMC_EVT_*`) [11].

Das Performance Monitoring (PM) ist ein Teilbereich der x86-Architektur, der einem häufigen Wandel zwischen verschiedenen Mikroarchitekturen unterworfen ist. So besitzt jede Mikroarchitektur einen unterschiedlichen Satz an verfügbaren Ereignissen. Für Intel Prozessoren befindet sich die vollständige Dokumentation im Reference Manual Volume 3 [11]. Auch herstellerübergreifend gibt es größere Unterschiede. So müssen die PMC auf CPUs von AMD häufig anders konfiguriert werden.

Im Rahmen der Bachelorarbeit wurde ein Treiber für diese PMC entwickelt[7] dessen Schnittstelle im Folgenden kurz vorgestellt wird.

`struct pmc_caps` Ist eine Datenstruktur, die Informationen über die Anzahl, Größe und Funktionen der verfügbaren Zähler enthält.

```
1 struct pmc_caps {
2   uint8_t  version;      // Architecural PM version
3   uint8_t  gp_count;     // Number of available GP PMCs
4   uint8_t  ff_count;     // Number of available FF PMCs
5   uint8_t  gp_width;     // Bit width of GP PMCs
6   uint8_t  ff_width;     // Bit width of FF PMCs
7   uint32_t arch_events; // Bit mask of supported events
```

[7] `metalsvm/arch/x86/kernel/pmc.c` und `metalsvm/arch/x86/include/asm/pmc.h`.

```
8     uint64_t msr;          // IA32_PERF_CAPABILITIES MSR
9 };
```

pmc_init() Initialisiert die PMCs und liefert ihre Eigenschaften über den Rückgabewert (`struct * pmc_caps`) zurück.

pmc_{gp,ff}_config(i, event, flags, mask) Konfiguiert einen Zähler. Der Parameter `flags` legt fest in welchen Privilegierungstufen und Prozessorzuständen das über `event` angegebene Ereignis gezählt werden soll (siehe Tabelle 5.4). Über das `mask` Argument können weitere Bedingungen an das Ereignis gestellt werden. Die Bedeutung dieser Masken ist dabei vom gewählten Ereignis abhängig.

pmc_{gp,ff}_start(i) Startet den Zähler mit Index `i`.

pmc_{gp,ff}_stop(i) Stoppt den Zähler mit Index `i`.

pmc_{gp,ff}_read,(i) Liest den Wert des Zählers mit Index `i` aus.

pmc_{gp,ff}_write(i, value) Setzt den Zähler mit Index `i` auf den Wert `value`.

pmc_{start,stop}_all() Startet bzw. stoppt alle Zähler gleichzeitig (atomar).

pmc_reset_all() Setzt alle Zähler auf den Wert 0 zurück.

Von besonderem Interesse sind die abgeschlossenen (engl. retired) Ereignisse. Sie treten nur auf, wenn die dazugehörige Instruktion oder der Zugriff auch vollständig ausgeführt wurde. Davon ausgenommen sind falsche Sprungvorhersagen oder spekulative Zugriffe, die zur Steigerung der Performance vom Prozessor ausgeführt wurden.

Auch QEMU signalisiert die Verfügbarkeit von PMCs für die in Tabelle 5.5 angegebenen CPUID-Werte. Bei genauerer Betrachtung wird deutlich, dass diese keine Werte zurückliefern. Vermutlich handelt es sich dabei um einen Fehler des Emulators. Selbst wenn die Emulation der PMC funktionieren würde, hätte sie keinen Nutzen. Die Emulation von PMCs macht keinen Sinn.

Architectural events	
UNHALTED_CORE_CLKS	UnHalted Core Cycles
INST_RET	Instruction Retired
LLC_REF	LLC Reference
LLC_MISS	LLC Misses
Non-architecural events	
DTLB_MISS_ANY	Memory accesses that missed the TLB
DTLB_MISS_LD	DTLB misses due to load operations
DTLB_MISS_L0_LD	L0 DTLB misses due to load operations
DTLB_MISS_ST	DTLB misses due to store operations
PAGE_WALK_COUNT	Number of page-walks executed
PAGE_WALK_CLKS	Duration of page-walks in core cycles
Precise & Retired events	
MEM_LOAD_RETIRED_L1D_MISS	Retired loads that miss the L1 data cache
MEM_LOAD_RETIRED_L2_MISS	Retired loads that miss the L2 cache
MEM_LOAD_RETIRED_DTLB_MISS	Retired loads that miss the DTLB

Tabelle 5.4.: Auszug der relevanten Performance-Monitoring-Ereignisse.

Eigenschaft		**CPUID**
PM Architektur Version	2	CPUID.0AH:EAX[7:0]
Anzahl der universellen PMCs	2	CPUID.0AH:EAX[15:8]
Anzahl der „Fixed Function“ PMCs	3	CPUID.0AH.EDX[4:0]
Bitbreite der universellen PMCs	40	CPUID.0AH:EAX[23:16]
Bitbreite der „Fixed Function“ PMCs	40	CPUID.0AH.EDX[12:5]

Tabelle 5.5.: Performance Monitoring Counters des Testsystems.

6. Benchmark und Tests

Zum Testen der virtuellen Speicherverwaltung von MetalSVM wurden kleine Testroutinen und Benchmarks genutzt. Die Tests[1] als auch der Benchmark[2] wurden dazu in einem eigenständigen Task des Kernels gestartet.

6.1. Tests

Getestet wurden sämtliche Funktionen, die in den Kapitel 2, 3 und 4 vorgestellt wurden. Neben dem Testsystem aus Kapitel 5.2 wurde dazu auch der Quick Emulator (QEMU) und VirtualBox eingesetzt. Bereits existierende Features wie die SMP-Unterstützung wurden auf Regressions-Fehler untersucht. Weiterhin wurde die Systemaufrufe `sbrk()`, `fork()` und `execve()` auf ihre Funktion geprüft. Da MetalSVM bisher noch keine User-Space-Prozesse im Longmode ausführen konnte, mussten dazu einige kleinere Portierungen ergänzt werden.

Besonders hervorzuheben ist dabei die AMD64-ABI, die gegenüber dem 32-Bit-Code eine neue Aufrufkonvention (engl. calling convention) verwendet [14]. Bei der 32 Bit-ABI werden die Funktionsargumente ausschließlich mit Hilfe des Stapelspeichers übergeben. Die aufrufende Funktion legt alle Argumente von rechts nach links auf den Stack. Die 64-Bit-ISA besitzt mit `R8` bis `R15` einen erweiterten Satz an Registern. Um die Parameterübergabe zu beschleunigen werden hier die ersten sechs Argumente mit Hilfe der Register (`RDI`, `RSI`, `RDX`, `RCX`, `R8` und `R9`) übertragen. Besitzt eine Funktion mehr als sechs Argumente werden die Restlichen wie bisher mit dem Stack übergeben. Diese neue Aufrufkonvention reduziert die Anzahl der Speicherzugriffe bei Funktionsaufrufen. Für die oben genannten Systemaufrufe ist eine Parameterübergabe über den Stack nicht möglich. Systemaufrufe gehen mit einem Wechsel in den Kernel einher und somit mit einem Wechsel des Stacks einher.

[1] `metalsvm/apps/memory.c`.
[2] `metalsvm/apps/membench.c`.

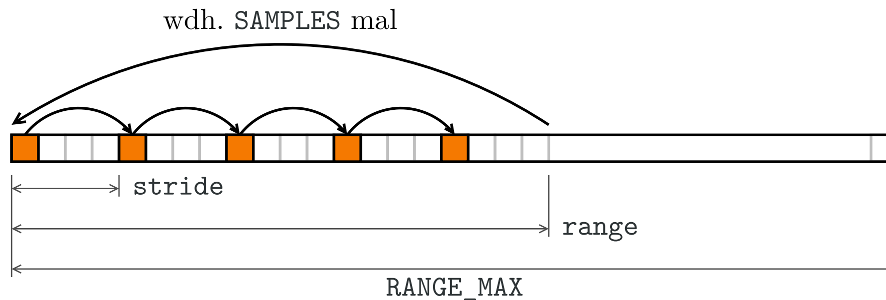


Abbildung 6.1.: Zugriffsmuster des Membench.

6.2. Membench

Um ein genaueres Bild der Speicherarchitektur des Testsystems zu erhalten wurde ein Speicher-Benchmark durchgeführt. *Membench* durchläuft einen kontinuierlichen Speicherbereich nach unterschiedlichen Mustern (siehe Abb. 6.1). Die Variablen `range` und `stride` werden durch zwei verschachtelte Schleifen variiert (siehe Listing 6.1).

```
#define RANGE_MAX (1 << 5)  /* 32 Byte */
#define RANGE_MAX (1 << 25) /* 32 MiB */
#define SAMPLES   (1000)

volatile char memory[RANGE_MAX], temp;
int range, stride, run, index;

for (range = RANGE_MIN; range < RANGE_MAX; range *= 2) {
  for (stride = 1; stride < range; stride *= 2) {
    for (run = 0; run < SAMPLES; run++) {
      for (index = 0; index < range; index += stride) {
        temp = memory[index]; /* read & write */
      }
    }
  }
}
```

Listing 6.1: Pseudocode des Membench Benchmarks.

Zur Auswertung des Benchmarks wurden die die in Kapitel 5.2.3 beschriebenen Performance Monitoring Counter (PMC) verwendet. Die Interrupts wurden deaktiviert, sowie Zwischenspeicher und TLB vor jedem Durchlauf geleert (engl. flush). Um weitere Seiteneffekte auszuschließen wurde jeder Testzyklus von einer

Speicher	**Takte**	**Zeit**
L1	$\approx$ 2,5	6,65 nS
L2	$\approx$ 4,2	11,12 nS
ASP	$\approx$ 47	125 nS

Tabelle 6.1.: Gemessene Latenzen der Speicherhierachie.

Aufwärm-Phase eingeleitet. Der damit bestimmte Overhead der Schleifendurchläufe wurde vom Ergebnis abgezogen.

Abbildungen 6.2 zeigt die benötigte Zeit pro Lesezugriff (Latenz). Die Zugriffszeiten lassen sich in drei Plateaus segmentieren (siehe Tabelle 6.1). Diese entsprechen der Speicherhierachie des Testsystems: Level 1 Cache (L1), Level 2 Cache (L2) und Arbeitsspeicher (ASP). Über die Grenzen dieser Plateaus lassen sich die Größe des L1/L2 Caches sowie die Größe einer Cache Line (CL) bestimmen. Die gemessenen Werte entsprechen dabei recht genau denjenigen, die für das Testsystem angegeben wurden (vgl. Tabelle 5.3). Aus der gleichen Grafik lässt sich so die Dauer einen Zugriffs auf den jeweiligen Speichertyp bestimmen. Tabelle 6.1 zeigt die Latenzen in Taktzyklen sowie Nanosekunden.

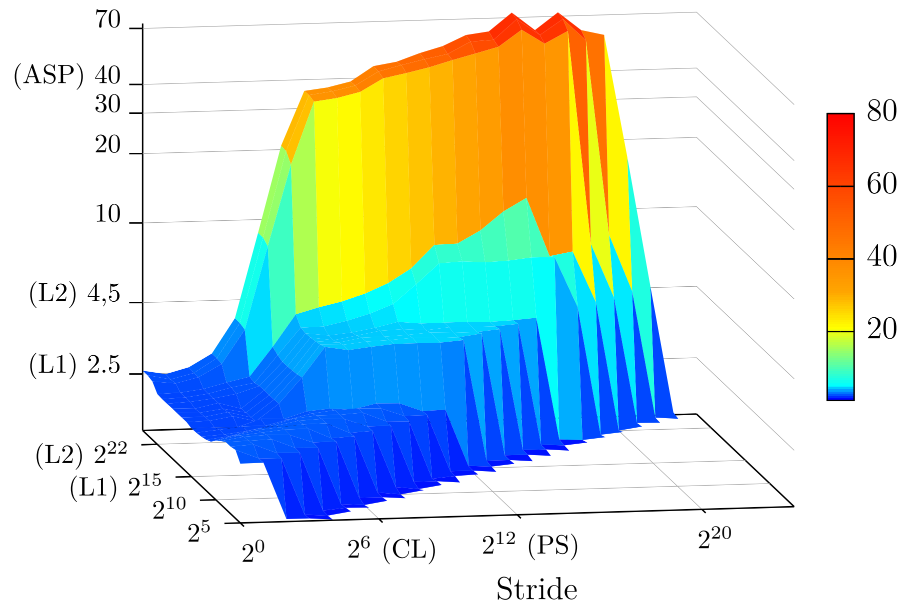
Mit Hilfe der PMC-Ereignisse `PAGE_WALK_COUNT` und `PAGE_WALK_CLKS` wurde die durchschnittliche Dauer eines Pagetable-Walks bestimmt. Diese liegt für einfache Durchläufe des Speichers bei knapp über 20 Zyklen. Ein normaler Pagetable-Walk greift im Longmode viermal auf den Speicher zu. Für jede Stufe einmal: PML4, PDPT, PGD, PGT. Die MMU nutzt für Speicherzugriffe den L2-Cache bzw. ASP. In der Regel befinden sich die Seitentabllen auch immer im L2-Cache. Ein Vergleich der Dauer von vier L2-Zugriffen ($4 \cdot 4,2 = 16,8$ Zyklen) mit der der Dauer eines Pagetable-Walks (≈ 20 Zyklen) erscheint plausibel.

Neben den Zugriffszeiten (Latenzen) werden auch die Anzahl der Cache-Misses auf den TLB, und L1/L2-Caches gemessen. Abbildung 6.3 zeigt die Anzahl der Cache-Misses pro Zugriff. Ein Wert von 1 deutet auf einen Miss bei jedem Zugriff hin. Auch diese Ergebnisse decken sich mit den bisher bekannten Werten.

Durch die Messung der TLB-Misses lassen sich auch Rückschlüsse auf die Größe des Puffers ziehen. Abbildung 6.4 zeigt wieder die Anzahl der Misses pro Zugriff. Übersteigt der *Stride* mit 4 KiB die Größe einer Seite (PS), steigen dort die Verfehlungen ab einer *Range* von 1 MiB rapide an. Mit Gleichung 6.1 lässt sich die Anzahl der Daten-TLB Einträge bestimmen (vgl. Tabelle 5.3):

$$\frac{1MiB}{4KiB} = \frac{2^{20}B}{2^{12}B} = 2^8 = 256 \tag{6.1}$$

Abbildung 6.4 zeigt auch, dass die TLB-Misses hauptsächlich dann auftreten, wenn zugleich auch der L2-Cache verfehlt wird (vgl. Abbildung 6.3). Die Auswirkungen der TLB-Misses werden dann durch die hohen Latenzen des Haupt-



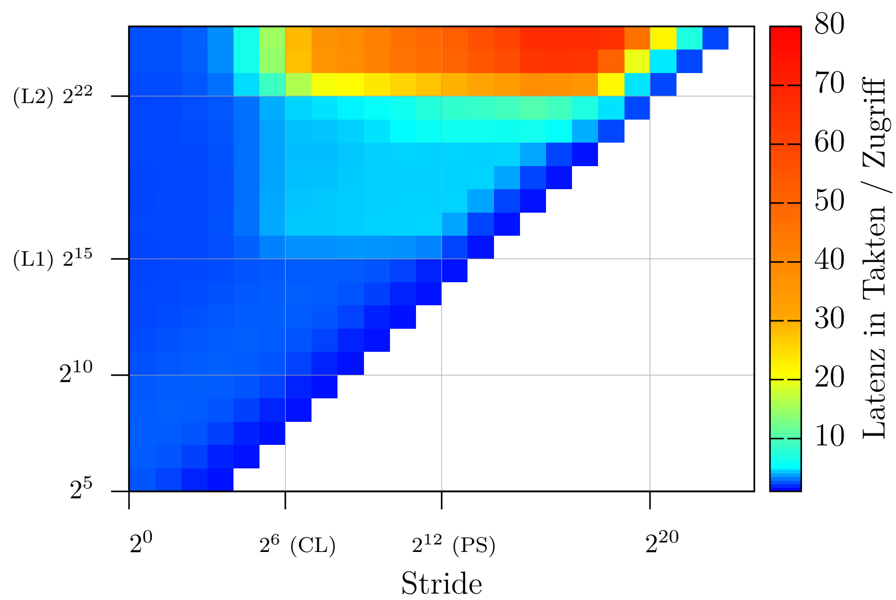
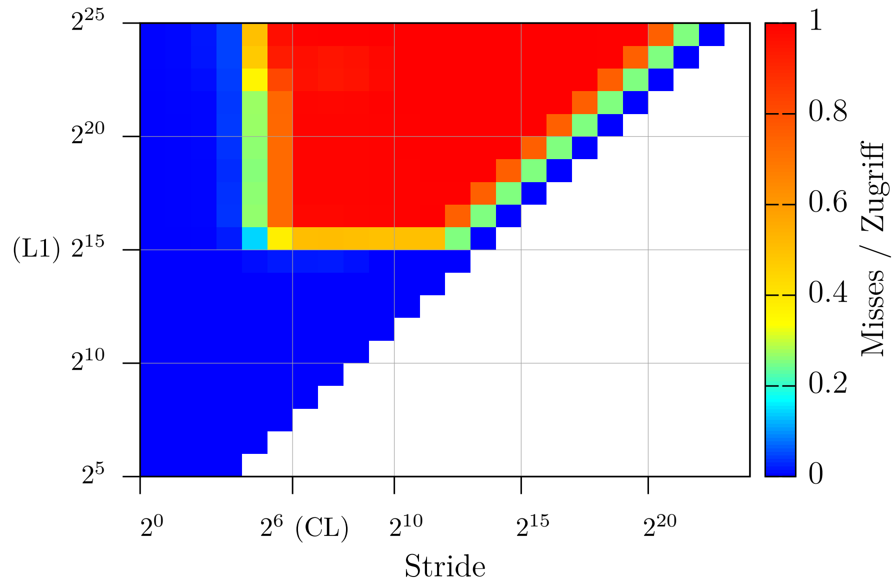
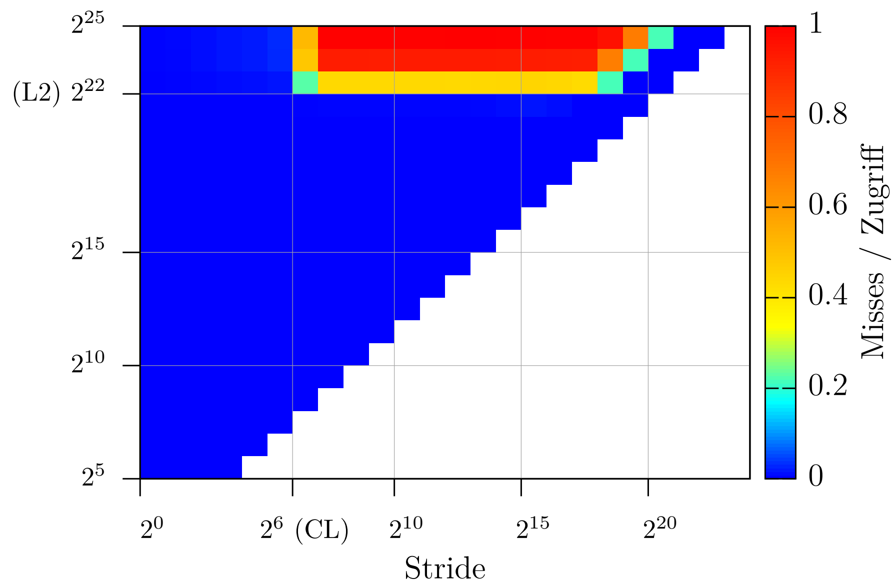


Abbildung 6.2.: Latenz des Membench.



(a) Level 1 Cache (L1).



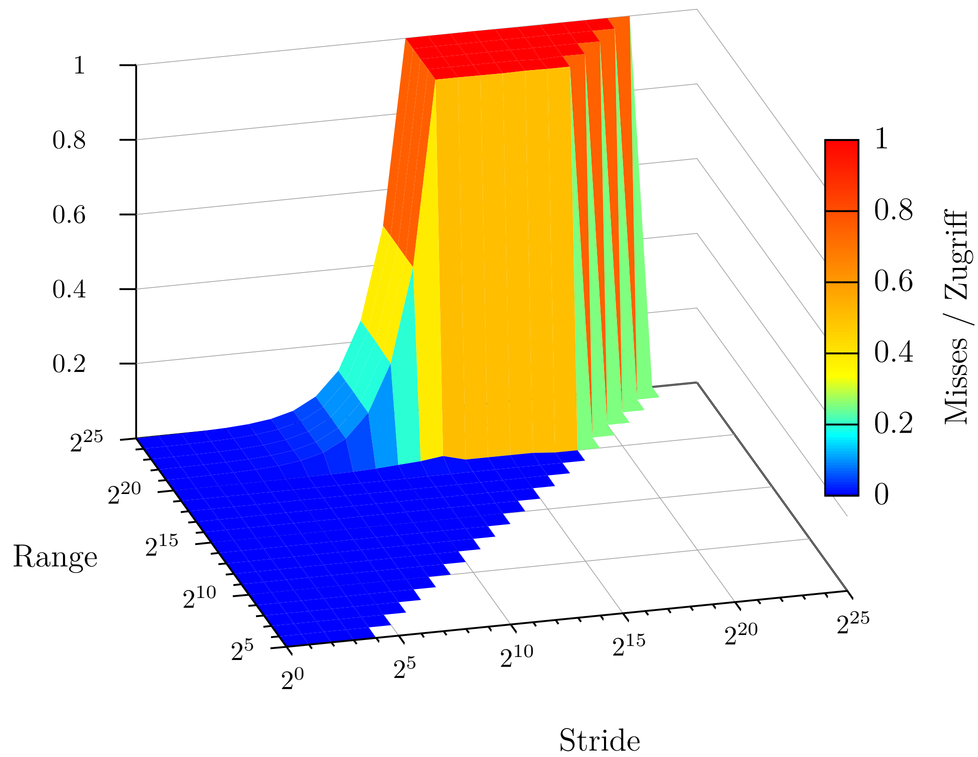
(b) Level 2 Cache (L2).

Abbildung 6.3.: Cache-Misses des Membench.

speicherzugriffs verdeckt. Die Größe des Translation Lookaside Buffers (TLBs) ist damit für das getestete Szenario optimal gewählt.

Weiterhin wäre es interessant den Einfluss von Hugepages auf die Anzahl der TLB-Misses und die Dauer eines Pagetable-Walks zu messen. Der verwendete Prozessor besitzt getrennte TLBs für 2/4 MiB und 1 GiB großen Hugepages. Diese TLBs besitzen eine Größe von 16/32 Einträgen und sind damit 8/16-mal kleiner als der TLB für 4 KiB-Seiten. Diese Tests konnten noch nicht durchgeführt werden, da MetalSVM noch keine finale Unterstützung für Hugepage besitzt.

Abbildung 6.5 zeigt die Anzahl der Takte, die für das Einblenden eines Bereichs benötigt werden. Auf der x-Achse ist die Größe des Bereichs in Seiten aufgetragen. Die benötigte Zeit hängt dabei annähernd linear von der Größe des Bereichs ab Alle 512 Seiten springt der Graph stufenförmig, da dann eine neue PGT erzeugt werden muss. Der eingeblendete Berreich ist dabei nicht an einem PGD ausgerichtet.



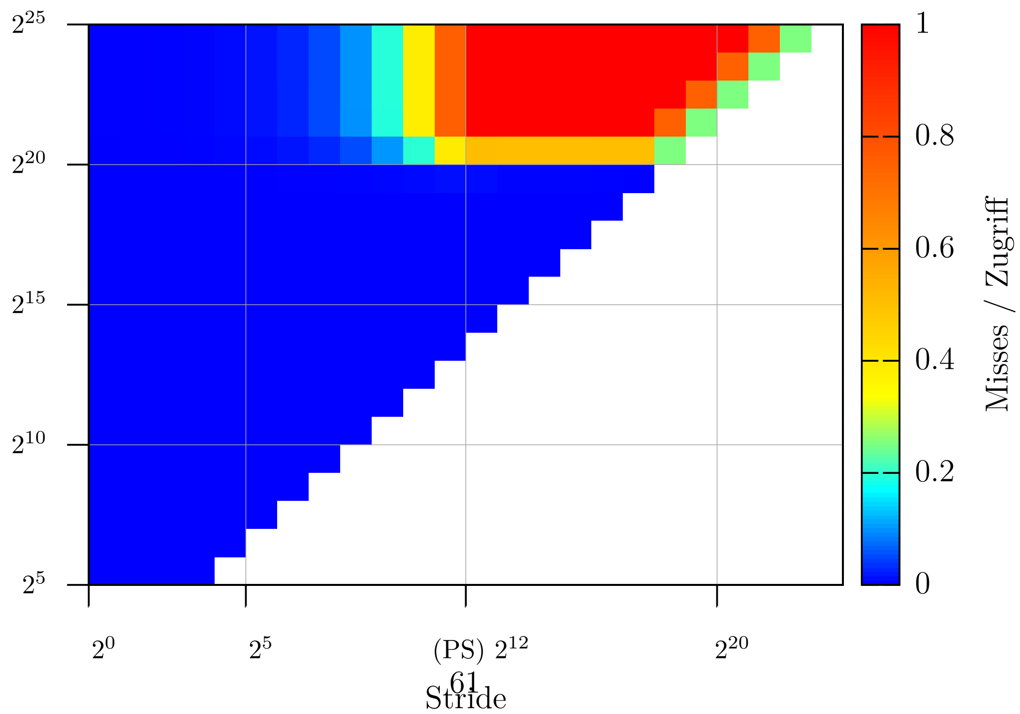


Abbildung 6.4.: Daten-TLB-Misses des Membench.

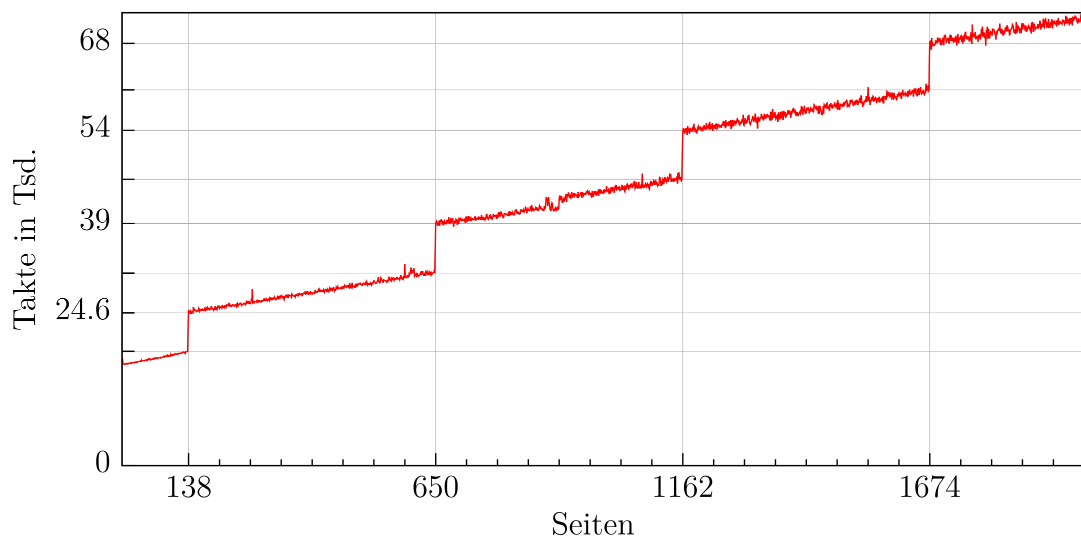


Abbildung 6.5.: Dauer von `map_region()` in Abhängigkeit der Seitenanzahl.

7. Zusammenfassung und Ausblick

Mit Hilfe der selbstreferenzierenden Seitentabellen konnte die Paging-Implementierung vereinfacht werden. Der dazu nötige Code wurde verkürzt und damit leichter verständlich. Auch andere Teile des Kernels wurden in diesem Hinblick vereinfacht und angepasst. So wurde der Kernel um eine dynamische Speicherverwaltung mittels `kmalloc()` bzw. `kfree()` erweitert. Bisher mussten dazu immer ganze Seiten allokiert werden. Mit dem neuen Buddysystem können auch Bruchteile einer Seite verwaltet und die Fragmentierung minimiert werden. Die bisher ausschließlich im User-Space vorhandenen Virtual Memory Areas (VMAs) finden jetzt auch im Kernel-Space ihren Einsatz. Sie ermöglichen es, schneller freie Speicherbereiche im virtuellen Adressraum zu finden ohne dazu die Seitentabellen durchsuchen zu müssen.

Ein Vergleich mit der Paging-Implementierung von Linux zeigt, dass MetalSVM beim Einblenden einzelner Seiten langsamer ist. Dazu wurden große Speicherbereiche im User-Space dynamisch angefordert und einfach durchlaufen. Sowohl Linux als auch MetalSVM blenden die dazugehörigen Seiten beim ersten Zugriff mit Hilfe des Pagefault-Handlers ein. Durch einen größeren Overhead im Handler sowie durch den Interrupt schneidet MetalSVM hier schlechter ab. Blendet man mehrere Seiten auf einmal ein, ist MetalSVM schneller. Es gilt daher noch diese Teile von MetalSVM weiter auf Performance zu optimieren.

Die aktuelle Version des Pagings wurde bisher nur auf 64-Bit-Systemen im Longmode getestet. Auch, wenn es sich um einen generischen Ansatz handelt, muss dieser noch für einen Einsatz auf 32-Bit-Systeme angepasst werden. Dazu wäre es vorteilhaft die Initialisierung der Prozessoren und Seitentabellen neu zu strukturieren. Zur Zeit nutzt MetalSVM dazu im Long- bzw. Protected-Mode getrennte Routinen[1]. Auch für Paging sind bisher zwei getrennte Varianten nötig gewesen[2]. Um die Duplizierung von Code zu vermeiden, wäre es wünschenswert auch diese Routinen zusammenzufassen. Dies gilt insbesondere für die Initialisierung weiterer Prozessoren (SMP).

Weiterhin könnte man die Einsatzmöglichkeit von selbstreferenzierenden Seitentabellen auf anderen Architekturen evaluieren. Als Beispiel hierfür würde die ARM-Architektur in Frage kommen. Für diese existiert bereits eine Portierung von

[1] `metalsvm/arch/x86/kernel/entry{32,64}.asm`.

[2] `metalsvm/arch/x86/kernel/mm/page{32,64}.c`.

MetalSVM. Die Abwärtskompatibilität besitzt bei ARM einen geringeren Stellenwert als bei Intel. So existieren für die verschiedenen Versionen der Architektur mehrere Varianten der Virtual Memory System Architecture (VMSA) (VMSAv5 bis VMSAv8). Mit *AArch64* wurde die Architektur um eine 64-Bit-Unterstützung ergänzt. Ähnlich wie bei der x86-Architektur wurde mit diesem Schritt das Paging auf vier Stufen ausgebaut. Neben MetalSVM wird am Lehrstuhl für Betriebssysteme (LfBS) mit *EduOS* auch an einem weiteren BS gearbeitet. Ursprünglich bildete EduOS die Grundlage für die Arbeit an MetalSVM. EduOS wird weiterhin gepflegt und in der Lehre eingesetzt. Eine Portierung dieser Arbeit auf EduOS wäre interessant.

A. Quellcode

Dieses Kapitel zeigt noch zwei weitere Funktionen der Paging-Implementierung, die aus Gründen ihres Umfangs nicht im Hauptteil aufgeführt wurden. Wie in Kapitel 2.3 wurden hier wieder Anweisungen zur Fehlerbehandlung und Synchronisation entfernt.

```
1  size_t map_region(size_t viraddr, size_t phyaddr,
2                    uint32_t npages, uint32_t flags) {
3    page_entry_t* first[PAGE_MAP_LEVELS];
4    page_entry_t* last[PAGE_MAP_LEVELS];
5
6    size_t bits = page_bits(flags);
7    size_t start = viraddr;
8    size_t end = start + npages * PAGE_SIZE;
9
10   int traverse(int level, page_entry_t* entry) {
11     page_entry_t* stop = entry + PAGE_MAP_ENTRIES;
12     for (; entry != stop; entry++) {
13       if (entry < last[level] && entry >= first[level]) {
14         if (level) { // PGD, PDPT, PML4..
15           if (*entry & PG_PRESENT) {
16             if ((flags & MAP_USER_SPACE)
17                  && !(*entry & PG_USER)) {
18               /* We are altering entries which cover
19                * the kernel. So before changing them we
20                * need to make a private copy for the task */
21               size_t phyaddr = get_page();
22               if (BUILTIN_EXPECT(!phyaddr, 0))
23                 return -ENOMEM;
24
25               copy_page(phyaddr, *entry & PAGE_MASK);
26               *entry = phyaddr | (*entry & ~PAGE_MASK);
27               *entry &= ~PG_GLOBAL;
28               *entry |= PG_USER;
29
30               /* We just need to flush the table itself.
```

```
31                 * TLB entries for the kernel remain valid
32                 * because we've not changed them. */
33                tlb_flush_one_page(entry_to_virt(entry, 0));
34              }
35            }
36            else {
37              /* Theres no page map table available which
38               * covers the region. Therefore we will create
39               * a new table. */
40              size_t phyaddr = get_page();
41              if (BUILTIN_EXPECT(!phyaddr, 0))
42                return -ENOMEM;
43
44              *entry = phyaddr | bits;
45
46              memset(get_child_entry(entry), 0x00,
                  ↪ PAGE_SIZE); // fill with zeros
47            }
48
49            // do "pre-order" traversal if no hugepage
50            if (!(*entry & PG_PSE)) {
51              int ret = traverse(level-1,
                  ↪ get_child_entry(entry));
52              if (ret < 0)
53                return ret;
54            }
55          }
56          else { // PGT
57            if ((*entry & PG_PRESENT) && !(flags & MAP_REMAP))
58              return -EINVAL;
59
60            *entry = phyaddr | bits;
61
62            if (flags & MAP_REMAP)
63              tlb_flush_one_page(entry_to_virt(entry, level));
64
65            phyaddr += PAGE_SIZE;
66          }
67        }
68      }
69
70      return 0;
71    }
```

```
72
73    // calc page tree boundaries
74    int i;
75    for (i=0; i<PAGE_MAP_LEVELS; i++) {
76      first[i] = virt_to_entry(start, i);
77      last[i] = virt_to_entry(end - 1, i) + 1; // exclusive
78    }
79
80    int ret = traverse(PAGE_MAP_LEVELS-1, current_map);
81
82    return (ret) ? 0 : viraddr;
83  }
```

Listing A.1: Die Funktion `map_region()`

```
1   int copy_page_map(task_t* new_task)
2   {
3     int traverse(int level,
4                  page_entry_t* src,
5                  page_entry_t* dest) {
6
7       page_entry_t* stop = src + PAGE_MAP_ENTRIES;
8       for (; src != stop; src++, dest++) {
9         if (*src & PG_PRESENT) {
10          if (*src & PG_USER) { // deep copy page frame
11            size_t phyaddr = get_page();
12
13            copy_page(phyaddr, *src & PAGE_MASK);
14            *dest = phyaddr | (*src & ~PAGE_MASK);
15
16            // do "pre-order" traversal
17            if (level && !(*src & PG_PSE))
18              int ret = traverse(level-1,
19                get_child_entry(src),
20                get_child_entry(dest)
21              );
22          }
23          else // shallow copy kernel table
24            *dest = *src;
25        }
26        else // table does not exists
27          *dest = 0;
```

```
28      }
29
30      return 0;
31    }
32
33    page_entry_t* src_virt = current_task->page_map;
34    page_entry_t* dest_virt = (page_entry_t*)
        ↪ palloc(PAGE_SIZE, MAP_KERNEL_SPACE);
35
36    size_t src_phys = virt_to_phys((size_t) src_virt);
37    size_t dest_phys = virt_to_phys((size_t) dest_virt);
38
39    // temporary map src and dest tables
40    current_map[PAGE_MAP_ENTRIES-2] =
41      (src_phys & PAGE_MASK) | PG_TABLE;
42    current_map[PAGE_MAP_ENTRIES-3] =
43      (dest_phys & PAGE_MASK) | PG_TABLE;
44
45    int ret = traverse(PAGE_MAP_LEVELS-1, src_map, dest_map);
46
47    // setup self reference for new table
48    dest_map[PAGE_MAP_ENTRIES-1] = dest_phys | PG_TABLE;
49
50    // unmap temporary tables
51    current_map[PAGE_MAP_ENTRIES-2] = 0;
52    current_map[PAGE_MAP_ENTRIES-3] = 0;
53
54    dest_map[PAGE_MAP_ENTRIES-2] = 0;
55    dest_map[PAGE_MAP_ENTRIES-3] = 0;
56
57    tlb_flush(); // ouch :(
58
59    new_task->page_map = dest_virt;
60
61    return ret;
62  }
```

Listing A.2: Die Funktion `copy_page_map()`

B. Kernel-Log

Die folgenden Ausgaben der Funktionen `page_dump()`, `vma_dump()` und `buddy_dump()` sind Teil der Testroutinen. Sie wurden zu Testzwecken implementiert und geben den aktuellen Zustand der einzelnen Teilbereiche der Speicherverwaltung dar.

```
start              -                 end           size flags
0x0000000000007000-0x0000000000008000            0x1000 -g---w
0x000000000009d000-0x000000000009e000            0x1000 -g---w
0x00000000000b8000-0x00000000000bb000            0x3000 -g---w
0x00000000000c0000-0x00000000000d3000           0x13000 -g---w
0x00000000000d4000-0x00000000000d7000            0x3000 -g---w
0x00000000000d9000-0x00000000000db000            0x2000 -g---w
0x00000000000f1000-0x00000000000f2000            0x1000 -g---w
0x00000000000ff000-0x0000000000100000            0x1000 -g---w
0x0000000000100000-0x00000000008c9000          0x7c9000 xg---w
0x00000000008c9000-0x0000000000979000           0xb0000 -g---w
0x0000000030000000-0x0000000030028000           0x28000 -g---w
0xffffff8000000000-0xffffff8000200000          0x200000 xg---w
0xffffffffc0000000-0xffffffffc0001000            0x1000 xg---w
0xfffffffffe00000-0xfffffffffe01000            0x1000 xg---w
0xfffffffffffff000-0000000000000000            0x1000 -----w
```

Listing B.1: Beispielausgabe `page_dump()`

```
Kernelspace VMAs:
0x7000 - 0x8000: size=1000, flags=rwx
0x9d000 - 0x9e000: size=1000, flags=r--
0xb8000 - 0xb9000: size=1000, flags=rw-
0xb9000 - 0xba000: size=1000, flags=rw-
0xba000 - 0xbb000: size=1000, flags=rw-
0xc0000 - 0xd3000: size=13000, flags=rw-
0xd4000 - 0xd7000: size=3000, flags=---
0xd7000 - 0xd9000: size=2000, flags=rw-
0xff000 - 0x100000: size=1000, flags=rwx
0x100000 - 0x8c9000: size=7c9000, flags=rwx
```

```
0x8c9000 - 0x9a1000: size=d8000, flags=rw-
Userspace VMAs:
0x40000000 - 0x40028000: size=28000, flags=rwx
0x80000000 - 0x8000a000: size=a000, flags=rw-
```

Listing B.2: Beispielausgabe `vma_dump()`

```
buddy_list[2] (exp=5, size=32 bytes):
  0xc1500 -> 0
buddy_list[3] (exp=6, size=64 bytes):
  0xc15c0 -> 0xc1600
  0xc1600 -> 0xc1640
  0xc1640 -> 0
buddy_list[4] (exp=7, size=128 bytes):
  0xc1680 -> 0
buddy_list[5] (exp=8, size=256 bytes):
  0xc1700 -> 0
buddy_list[8] (exp=11, size=2048 bytes):
  0xc1800 -> 0
buddy_list[11] (exp=14, size=16384 bytes):
  0xcc000 -> 0
buddy_list[15] (exp=18, size=262144 bytes):
  0x979000 -> 0
```

Listing B.3: Beispielausgabe `buddy_dump()`

Das folgende Kernel-Log zeigt die Ausgaben des Kernels während des Bootvorgangs. Zeilen, die die Speicherverwaltung betreffen, wurden dabei farbig hervorgehoben (Paging, VMA, Heap, Allgemeines).

```
This is MetalSVM 0.901 Build 20140503, 164220
Paging features: PSE (2/4Mb) PAE PGE PAT PSE36 NX LM
Physical adress-width: 40 bits
Linear adress-width: 48 bits
Found and initialized FPU!
MetalSVM is running on a hypervisor!
Hypervisor Vendor Id: ?
Maximum input value for hypervisor CPUID info: 0x121
Found MP config table at 0xf1800
System uses Multiprocessing Specification 1.4
MP features 1: 0
```

```
12 Found IOAPIC at 0xfec00000
13 Found 1 cores
14 Found APIC at 0xfee00000
15 Maximum LVT Entry: 0x5
16 APIC Version: 0x11
17 Boot processor 0 (ID 0)
18 map_region: map 1 pages from 0x8c4000 to 0x8c4000 with
     ↪ flags: 0x1000
19 Map module tools/initrd.img at 0x8c5000 (176 pages)
20 map_region: map 176 pages from 0x8c5000 to 0x8c5000 with
     ↪ flags: 0x1000
21 Mapped LAPIC at 0xb9000
22 map_region: map 1 pages from 0xba000 to 0xfec00000 with
     ↪ flags: 0x20
23 Mapped IOAPIC at 0xba000
24 map_region: map 1 pages from 0xf1000 to 0xf1000 with flags:
     ↪ 0x1020
25 IOAPIC version: 0x11
26 Max Redirection Entry: 23
27 vma_add: start = 0x100000, end = 0x8c4000, flags = 0xf
28 palloc(65536) (16 pages)
29 vma_alloc: size = 0x10000, flags = 0xb
30 get_pages: ret 0xa000, i = 10, j = 16, npages = 16
31 map_region: map 16 pages from 0xc0000 to 0xa000 with flags:
     ↪ 0
32 kmalloc(48) = 0xc0008
33 vma_add: start = 0xf1000, end = 0xf2000, flags = 0xf
34 kmalloc(48) = 0xc0048*]
35 vma_add: start = 0xb9000, end = 0xba000, flags = 0x3
36 kmalloc(48) = 0xc0088
37 vma_add: start = 0xba000, end = 0xbb000, flags = 0x3
38 kmalloc(48) = 0xc00c8
39 vma_add: start = 0xb8000, end = 0xb9000, flags = 0x3
40 kmalloc(48) = 0xc0108
41 vma_add: start = 0x9000, end = 0xa000, flags = 0x9
42 kmalloc(48) = 0xc0148
43 vma_add: start = 0x9000, end = 0xa000, flags = 0x9
44 vma_add: start = 0x8c4000, end = 0x8c5000, flags = 0x9
45 kmalloc(48) = 0xc0188
46 vma_add: start = 0x8c5000, end = 0x975000, flags = 0xb
47 kmalloc(48) = 0xc01c8
48 kmalloc(4360) = 0xc2008
49 kmalloc(208) = 0xc0208
```

```
50 kmalloc(4360) = 0xc4008
51 kmalloc(208) = 0xc0308
52 kmalloc(4360) = 0xc600
53 kmalloc(208) = 0xc1008
54
55   [...]
56
57 kmalloc(208) = 0xc1308
58 kmalloc(208) = 0xc1408
59 Kernel starts at 0x100000 and ends at 0x8c3e88
60 APIC calibration determines an ICR of 0x98078d
61 Processor frequency: 2261 MHz
62 Total memory: 127 MBytes
63 Current allocated memory: 8728 KBytes
64 Current available memory: 119 MBytes
65 kmalloc(24) = 0xc1508
66 palloc(4096) (1 pages)
67 vma_alloc: size = 0x1000, flags = 0xb
68 get_pages: ret 0x1a000, i = 26, j = 1, npages = 1
69 map_region: map 1 pages from 0xd0000 to 0x1a000 with flags:
     ↪ 0
70 copy_page_map: copy = 0, src = 0x27c000 (0x27c000,
     ↪ 0xfffffffffffe000), dest = 0xd0000 (0x1a000,
     ↪ 0xffffffffffffd000)
71 palloc(8192) (2 pages)
72 vma_alloc: size = 0x2000, flags = 0xb
73 get_pages: ret 0x1b000, i = 27, j = 2, npages = 2
74 map_region: map 2 pages from 0xd1000 to 0x1b000 with flags:
     ↪ 0
75
76   [...]
77
78 palloc(4096) (1 pages)
79 vma_alloc: size = 0x1000, flags = 0xb
80 get_pages: ret 0x1d000, i = 29, j = 1, npages = 1
81 map_region: map 1 pages from 0xd3000 to 0x1d000 with flags:
     ↪ 0
82 copy_page_map: copy = 0, src = 0x27c000 (0x27c000,
     ↪ 0xfffffffffffe000), dest = 0xd3000 (0x1d000,
     ↪ 0xffffffffffffd000)
83 palloc(8192) (2 pages)
84 vma_alloc: size = 0x2000, flags = 0xb
85 get_pages: ret 0x1e000, i = 30, j = 2, npages = 2
```

```
86  map_region: map 2 pages from 0xd4000 to 0x1e000 with flags:
      ↪ 0
87  ======= Starting membench
88  PMC architecural version: 0
89  There are 0 general purpose PMCs (0 bit wide) available
90  There are 0 fixed function PMCs (0 bit wide) available
91  kfree(791816)
92  Terminate task: 1, return value 0
93  drop_page_map: task = 1
94  user_usage: 0 (task = 1)
95  put_pages: phyaddr=0xd0000, npages = 1, ret = 1
96  put_pages: phyaddr=0x1b000, npages = 1, ret = 1
97  put_pages: phyaddr=0x1c000, npages = 1, ret = 1
98  unmap_region: unmap 2 pages from 0xd1000
99  vma_free: start = 0xd1000, end = 0xd3000
100 kmalloc(48) = 0xc1548
101 palloc(8388608) (2048 pages)
102 vma_alloc: size = 0x800000, flags = 0xb
103 get_pages: ret 0x975000, i = 2421, j = 2048, npages = 2048
104 map_region: map 2048 pages from 0x975000 to 0x975000 with
      ↪ flags: 0
105 Allocated test memory: 0x800000 bytes at 0x975000
106
107   [...]
```

C. Werkzeuge

C.1. GDB Longmode Patch

Wie in Kapitel 5.1.3 erläutert, musste der GDB für die Fehlersuche angepasst werden. Bei einem Wechsel der Architektur stürzt der Debugger sonst mit der folgenden Fehlermeldung ab: „remote 'g' packet reply is too long“. Dieses „g packet“ enthält den aktuellen Zustand der Register. Wechselt die CPU während der Initialisierung vom Protected-Mode in den Longmode müssen die Register mit einer doppelten Breite übertragen werden. Darauf ist GDB jedoch nicht vorbereitet und erzeugt die oben angegebene Meldung. Der Patch C.1 stellt einen Workaround[1] für dieses Problem dar. Erkennt der GDB ein längeres „g packet“ impliziert es damit einen Wechsel in den Longmode und passt die Interpretation des Pakets entsprechend an. Zusätzlich muss die Architektur manuell über die GDB-Konsole gewechselt werden: `set architecture i386:x86-64`.

```
1  --- gdb/remote.c.orig 2013-10-15 11:36:30.287817739 +0200
2  +++ gdb/remote.c  2013-10-15 11:54:28.225162932 +0200
3  @@ -6109,9 +6109,24 @@ process_g_packet (struct regcache *regca
4  
5     buf_len = strlen (rs->buf);
6  
7  -  /* Further sanity checks, with knowledge of the architecture.  */
8  -  if (buf_len > 2 * rsa->sizeof_g_packet)
9  -    error (_("Remote 'g' packet reply is too long: %s"), rs->buf);
10 +  /* Further sanity checks, with knowledge of the architecture.
11 +     By: z0rr0 from
    ↪ http://forum.osdev.org/viewtopic.php?f=13&p=177644 */
12 +  if (buf_len > 2 * rsa->sizeof_g_packet) {
13 +    rsa->sizeof_g_packet = buf_len;
14 +
15 +    warning (_("remote 'g' packet reply is too long; buffer
    ↪ resized"));
16 +    warning (_("  probable longmode entered"));
17 +
18 +    for (i = 0; i < gdbarch_num_regs (gdbarch); i++)
19 +    {
```

[1] `http://forum.osdev.org/viewtopic.php?f=13&p=177644`.

```
20 +        if (rsa->regs[i].pnum == -1)
21 +          continue;
22 +
23 +        if (rsa->regs[i].offset >= rsa->sizeof_g_packet)
24 +          rsa->regs[i].in_g_packet = 0;
25 +        else
26 +          rsa->regs[i].in_g_packet = 1;
27 +      }
28 +    }
29
30     /* Save the size of the packet sent to us by the target.  It is
         ↪ used
31        as a heuristic when determining the max size of packets that
          ↪ the
```

C.2. iPXE Konfiguration

Für einen schnellen Entwicklungszyklus wurde der MetalSVM-Kernel über das Netzwerk gebootet. Dazu wurde auf dem Entwicklungsrechner ein einfacher HTTP-Server gestartet. Der Testrechner besitzt eine Realtek-RT8139-kompatible Netzwerkkarte, die über einen Boot-ROM verfügt. Auf diesen Speicher wurde ein Bootloader namens „iPXE“ geflasht.

Die folgenden zwei Auszüge stellen die Konfiguration des Bootloaders dar, die verkettet (engl. chained) geladen werden. Der erste Auszug wird bei der Kompilation von iPXE direkt mit in das ROM-Abbild mit eingebaut. Dieser lädt im Anschluss ein weiteres Skript, das den Kernel lädt. Durch diese Verkettung kann auch ohne erneutes Flashen des Boot-ROMs die Konfiguration geändert werden.

```
1 #!ipxe
2
3 set server 134.130.62.174:8080
4 set script script.ipxe
5
6 dhcp
7 chain http://${server}/${script}
```

```
1 #!ipxe
2
3 # iPXE (http://ipxe.org) is an opensource network boot
```

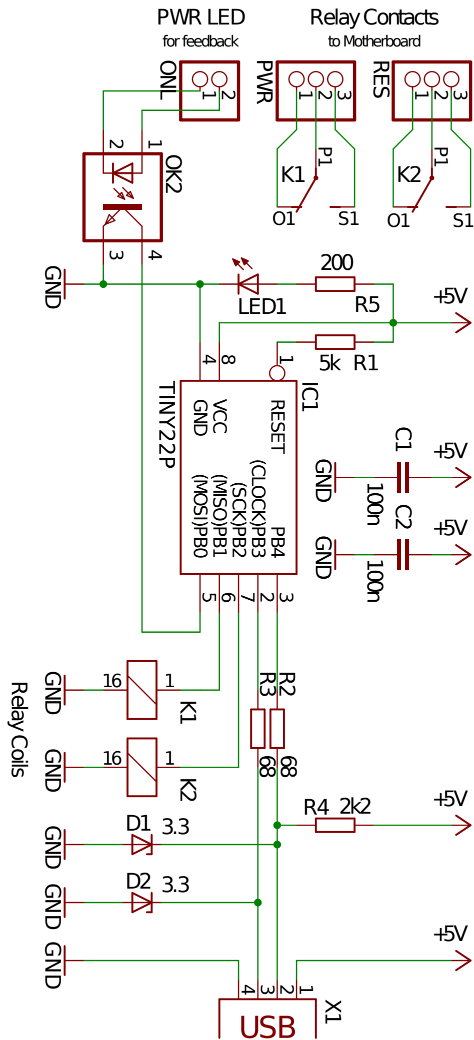
```
    ↪ firmware.
 4  # It provides a full PXE implementation enhanced with
 5  # additional features such as booting from HTTP, FTP,
 6  # iSCSI SAN, Fibre Channel SAN, Wireless, WAN or Infiniband
 7  #
 8  # We use it to rapidly compile & debug MetalSVM on real
    ↪ hardware.
 9
10  set server 134.130.62.174:8080
11
12  dhcp
13
14  kernel http://${server}/metalsvm.elf
15  module http://${server}/tools/initrd.img
16
17  boot
```

C.3. HWreset

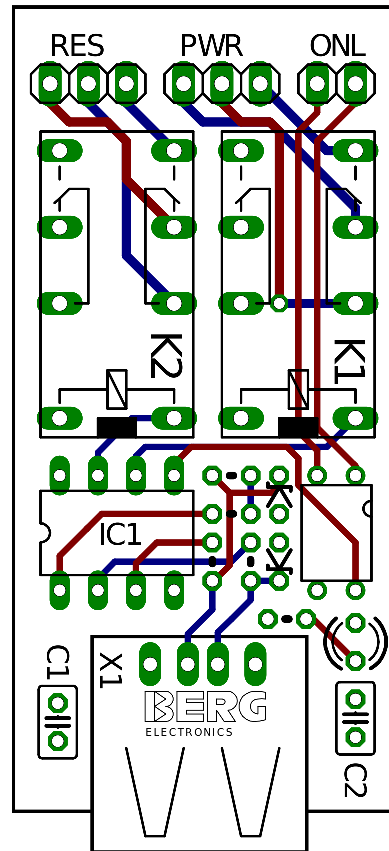
Während der Arbeit an MetalSVM ist mit HWreset[2] ein kleiner USB-Stick zum Zurücksetzen des Testrechners entstanden. HWreset wird über USB an den Entwicklungsrechner angeschlossen und kann mittels Kleinsignalrelais den Testrechner neustarten und testen ob dieser zur Zeit in Betrieb ist. In Verbindung mit iPXE (siehe Abschnitt C.2) und der Ein- / Ausgabe über einen RS232-Port kann so das Testen mit Makefiles vollständig automatisiert werden. Das Target `hwdebug` im Makefile[3] führt folgende Schritte aus:

1. Kernel kompilieren (`make metalsvm.elf`)
2. HTTP Server starten (`mongoose -p 8080`)
3. Testsystem resetten (`hwreset push reset 500`)
4. Testsystem lädt Kernel und Initiale Ramdisk via iPXE über das Netzwerk
5. Ausgaben mit Terminal Emulator mitschneiden (`cu -s 115200 -l /dev/ttyUSB0`)

[2] `https://github.com/stv0g/hardware/tree/master/HWreset`.
[3] `metalsvm/Makefile`.



(a) Schaltplan



(b) Platinenlayout

Abbildung C.1.: Schaltplan und Layout von HWreset.

D. Alter Ansatz

Abbildung D.1 zeigt den bisher von MetalSVM genutzten Ansatz zum Einblenden der Seitentabellen.

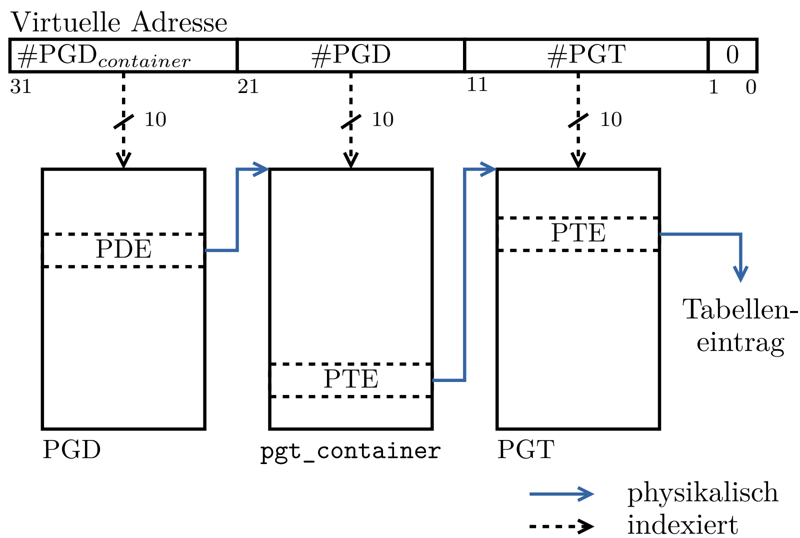


Abbildung D.1.: Alter Ansatz des Pagings von MetalSVM im Protected-Mode.

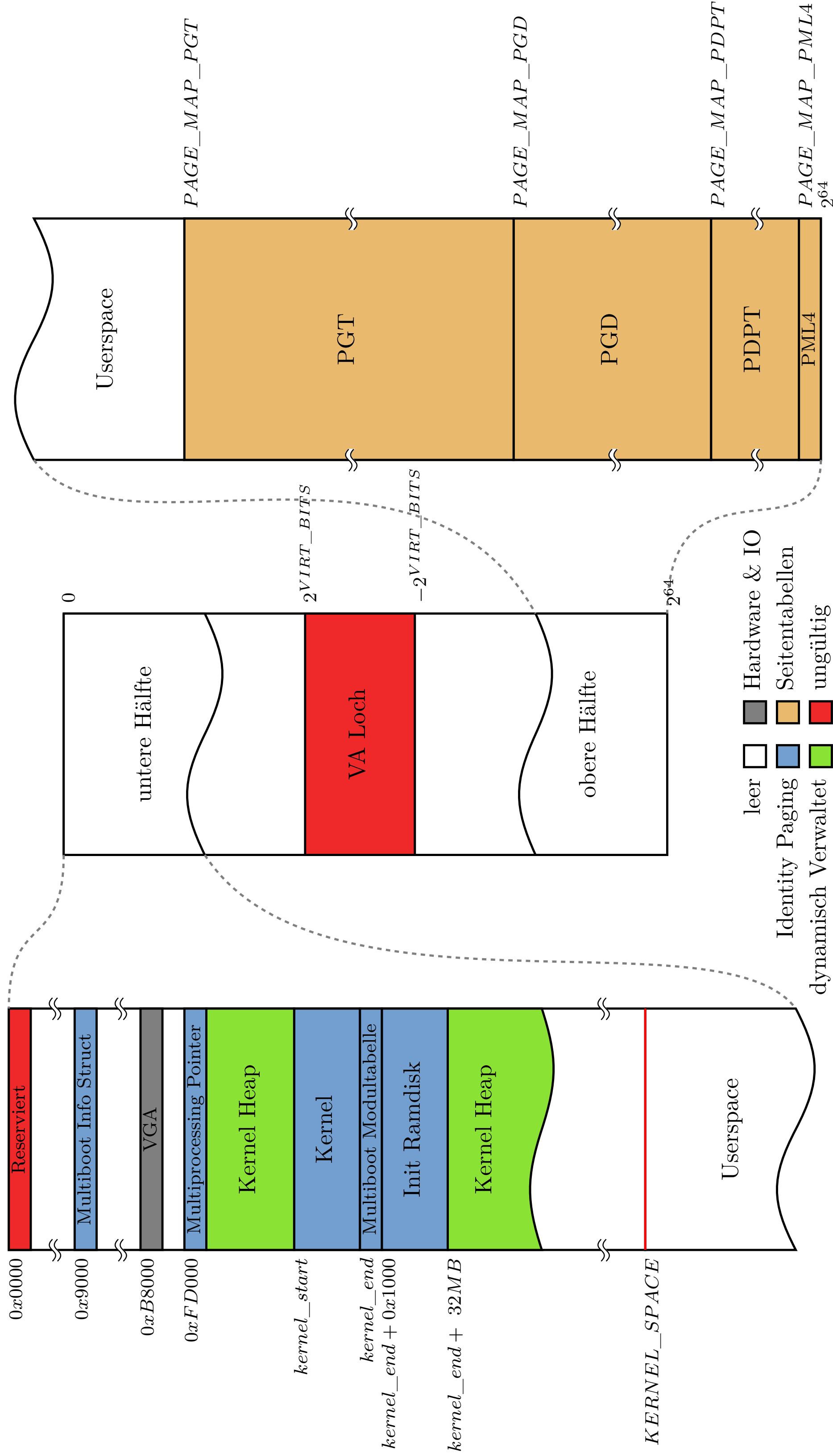


Abbildung D.2.: Speicherabbild von MetalSVM im Longmode.

Literaturverzeichnis

[1] ARM Limited (Hrsg.): *ARM Architecture Reference Manual.* ARMv7-A and ARMv7-R. Cambridge, England: ARM Limited, Juli 2012. `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0406b/index.html`, Abruf: Februar 2014

[2] ARM Limited (Hrsg.): *ARM Architecture Reference Manual.* ARMv8-A, beta. Cambridge, England: ARM Limited, Oktober 2013. `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.set.architecture/index.html`, Abruf: Februar 2014. – mit Errata Markierungen

[3] Compaq Computer Corporation (Hrsg.): *Alpha Architecture Reference Manual.* 4. Houston, TX, USA: Compaq Computer Corporation, Januar 2002

[4] Dave Probert: Windows Kernel Architecture Internals. Version: April 2010. `http://research.microsoft.com/en-us/um/redmond/events/wincore2010/Dave_Probert_1.pdf`, Abruf: April 2014. Microsoft

[5] Digital Equipment Corporation (Hrsg.): *VAX-11 Architecture Reference Manual.* Revision 6.1. Maynard, MA, USA: Digital Equipment Corporation, Juni 1981

[6] Freescale Semiconductor, Inc. (Hrsg.): *Programming Environments Manual for 32-Bit Implementations of the PowerPCTM Architecture.* Revision 3. Chandler, AZ, USA: Freescale Semiconductor, Inc., September 2005

[7] Fujitsu Limited (Hrsg.): *SPARC JPS1 Implementation Supplement: Fujitsu SPARC64 V.* Release 1.0. Kamikodanaka, Japan: Fujitsu Limited, Juli 2002

[8] Gorman, M. : *Unterstanding The Linux Virtual Memory Manager* `https://www.kernel.org/doc/gorman/pdf/understand.pdf`

[9] Intel Corporation (Hrsg.): *Intel 64 and IA-32 Architectures Optimization Reference Manual.* Santa Klara, CA, USA: Intel Corporation, März 2014. `http://www.intel.com/content/dam/www/public/us/en/documents/`

manuals/64-ia-32-architectures-optimization-manual.pdf, Abruf: April 2014

[10] Intel Corporation (Hrsg.): *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture*. Santa Klara, CA, USA: Intel Corporation, Februar 2014. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-1-manual.pdf, Abruf: Februar 2014

[11] Intel Corporation (Hrsg.): *Intel 64 and IA-32 Architectures Software Developer's Manual, Volumes 3A, 3B & 3C: System Programming Guide*. Santa Klara, CA, USA: Intel Corporation, Februar 2014. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-system-programming-manual-325384.pdf, Abruf: Februar 2014

[12] International Business Machines Corporation (Hrsg.): *PowerPC Microprocessor Family: The Programming Environments Manual for 64-bit Microprocessors*. Version 3.0. New York, NY, USA: International Business Machines Corporation, Juli 2005

[13] Knuth, D. E.: *The Art of Computer Programming Volume 1: Fundamental Algorithms*. 3 A. Massachusetts, USA : Addison-Wesley, 1997. – ISBN 978–0201896831

[14] Matz, M. ; Hubicka, J. ; Jaeger, A. ; Mitchell, M. : *System V Application Binary Interface*. Draft Version 0.99.6, Oktober 2013. http://www.x86-64.org/documentation/abi.pdf, Abruf: Februar 2014

[15] MIPS Technologies, Inc. (Hrsg.): *MIPS® Architecture For Programmers Volume III: The MIPS32® and microMIPS32TM Privileged Resource Architecture*. Revision 5.03. Sunyvale, CA, USA: MIPS Technologies, Inc., September 2013

[16] MIPS Technologies, Inc. (Hrsg.): *MIPS® Architecture For Programmers Volume III: The MIPS64® and microMIPS64TM Privileged Resource Architecture*. Revision 5.03. Sunyvale, CA, USA: MIPS Technologies, Inc., September 2013

[17] Okuji, Y. K. ; Ford, B. ; Boleyn, E. S. ; Ishiguro, K. : *The Multiboot Specification*. Version 0.6.96. Boston, MA, USA: Free Software Foundation, 2009. http://www.gnu.org/software/grub/manual/multiboot/multiboot.pdf, Abruf: Februar 2014

[18] Reble, P. ; Galowicz, J. ; Lankes, S. ; Bemmerl, T. : Efficient Implementation of the bare-metal Hypervisor MetalSVM for the SCC. In: *Proceedings of the 6th Many-core Applications Research Community (MARC) Symposium.* – ISBN 978–2725700168, 59–65

[19] Shen, K. K. ; Peterson, J. L.: A Weighted Buddy Method for Dynamic Storage Allocation. In: *Commun. ACM* 17 (1974), Oktober, Nr. 10, S. 558–562

[20] SPARC International Inc. (Hrsg.): *The SPARC Architecture Manual.* Version 8. San Jose, CA, USA: SPARC International Inc., 1992

[21] SPARC International Inc. (Hrsg.): *The SPARC Architecture Manual.* Version 9. San Jose, CA, USA: SPARC International Inc., 1994

[22] Sun Microsystems Computer Corporation (Hrsg.): *The SuperSPARC Microprocessor: Technical Whitepaper.* Mountain View, CA, USA: Sun Microsystems Computer Corporation, 1992

[23] Sun Microsystems Computer Corporation (Hrsg.): *UltraSPARC Architecture 2005.* Draft D0.9.2. Santa Klara, CA, USA: Sun Microsystems Computer Corporation, Juni 2008

[24] Tanenbaum, A. S.: *Modern Operating Systems.* 3. Pearson International, 2007. – ISBN 978–0138134594

Abkürzungsverzeichnis

ABI Application Binary Interface

AMD64 AMDs Bezeichnung für x86-64

APIC Advanced Programmable Interrupt Controller

ASP Arbeitsspeicher

BS Betriebssystem

CISC Complex instruction set computing

CL Cache Line

CPU Central Processing Unit

GDB GNU Debugger

GDT Global Descriptor Table

GRUB Grand Unified Bootloader

HGSP Hintergrundspeicher

HPC High Performance Computing

IA-32e „Intel Architecture with 32-Bit and Longmode Extension“

IA-32 „Intel Architecture with 32-Bit“

IA-64 „Intel Architecture with 64-Bit (Itanium)“

InitRd Init Ramdisk

I/O-APIC Input/Output APIC

IPT Inverted Page Table

ISA Instruction Set Architecture

KVM Kernel-based Virtual Machine

L1 Level 1 Cache

L2 Level 2 Cache

LAPIC Local APIC

LDT Local Descriptor Table

LfBS Lehrstuhl für Betriebssysteme

libc C-Standard-Bibliothek

LWK Lightweight Kernel Operating System

MIC Many Integrated Core Architecture

MM Memory Management

MMU Memory Management Unit

MSR Model-specific Register

NUMA Non-Uniform Memory Access

PAE Physical Address Extension

PA Physikalischer Adressraum

PCB Printed Circuit Board

PCI Peripheral Component Interconnect

PFN Page Frame Number

PID Process Identifier

PMC Performance Monitoring Counter

PMU Performance Monitoring Unit

PSE Page Size Extension

PXE Preboot Execution Environment

QEMU Quick Emulator

RAM Random-access Memory

ROM Read-only Memory

RDMA Remote direct memory access

RISC Reduced instruction set computing

SCC Single-chip Cloud Computer

SEXT Sign Extension

SMP Symmetric Multiprocessing

SVM Shared Virtual Machine

UART Universal Asynchronous Receiver Transmitter

TLB Translation Lookaside Buffer

VA Virtueller Adressraum

VGA Video Graphics Array

VMA Virtual Memory Area

VMSA Virtual Memory System Architecture

VPN Virtual Page Number

x86-64 Intel 80386 kompatible Architektur mit 64 Bit-Erweiterung

x86 Intel 80386 kompatible Architektur

Hersteller

AMD Advanced Micro Devices

ARM ARM Limited

DEC Digital Equipment Corporation

IBM International Business Machines Corporation

VIA VIA Technologies

Paging Tabellen und Einträge

PML4 Page Map Level 4

PML4E Page Map Level 4 Entry

PDPT Page Directory Pointer Table

PDPTE Page Directory Pointer Table Entry

PGD Page Directory

PDE Page Directory Entry

PGT Page Table

PTE Page Table Entry

PF Page Frame

Paging Flags

P Present

R/W Read/Write

U/S User/Supervisor

PWT Page-level write-through

PCD Page-level cache-disable

A Accessed

D Dirty

PS Page Size

PAT Page Attribute Table

G Global

XD Execute-disable

x86 Register und Bits

CR0 Control Register 0

CR3 Control Register 3

CR4 Control Register 4

MTRR Memory Type Range Register

EFER IA-32 Extended Feature Enable Register