

# Übung zu Betriebssystemtechnik

## Aufgabe 3: Paging in STUBSMI

---

16. Mai 2024

Dustin Nguyen, Maximilian Ott & Phillip Raffeck

Lehrstuhl für Informatik 4  
Friedrich-Alexander-Universität Erlangen-Nürnberg



**Lehrstuhl für Informatik 4**  
Systemsoftware



**Friedrich-Alexander-Universität**  
Technische Fakultät

## Nachtrag: Systemaufrufbehandlung

```
extern "C" size_t syscall_handler(size_t p1, size_t p2, size_t p3,
                                size_t p4, size_t p5, size_t sysnum) {
    // Fast path for benchmarking
    if (sysnum == SYSCALL_NOP)
        return 0;

    // Enter epilogue level
    Guarded section;
    // Enable Interrupts
    Core::Interrupt::enable();

    // Call syscall skeleton
    switch(sysnum) {
        case SYSCALL_WRITE:
            return Skeleton::write(static_cast<int>(p1), reinterpret_cast<void*>(p2), p3);
        // ...
        default:
            DBG << "Unknown SYSCALL " << sysnum << endl;
            return static_cast<size_t>(-1);
    }
    return 0;
}
```

**Motivation für die nächsten  
beiden Aufgaben (3 & 4)**

---

# Status Quo

0xffff ffff



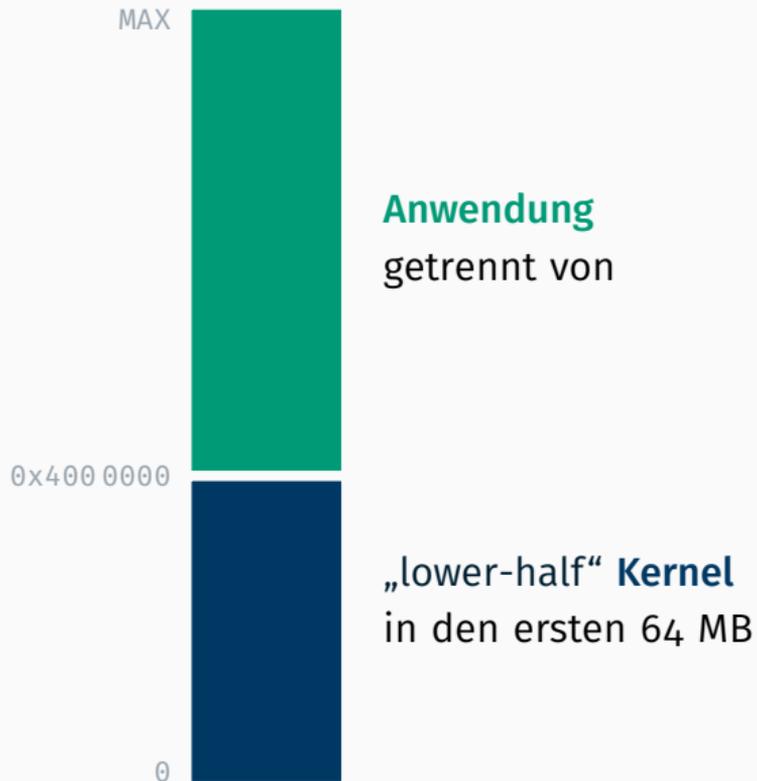
0

**Kernel** und **Anwendung**

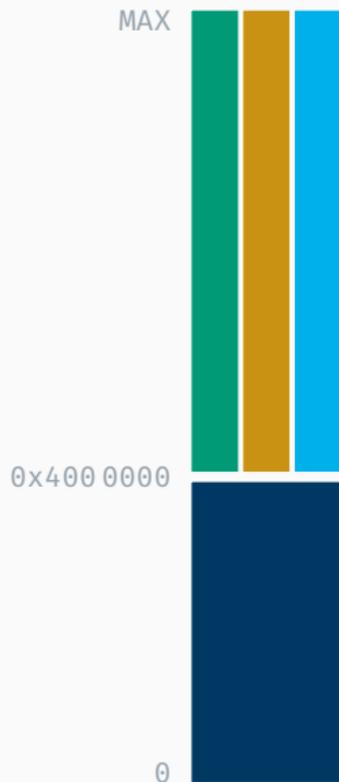
ineinander verwoben

(sowohl bei der Entwicklung als auch Ausführung)

# Nach Aufgabe 4



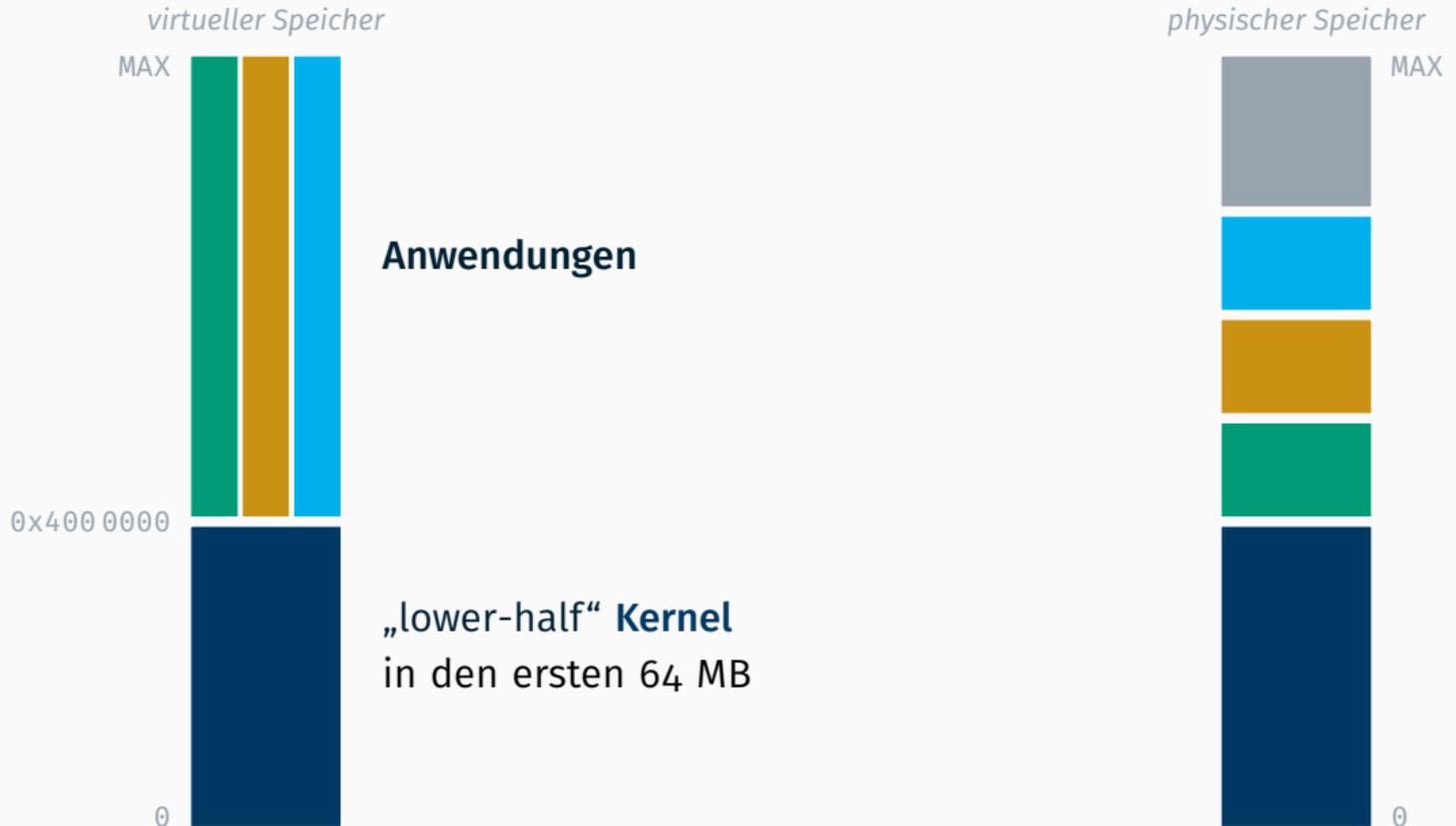
# Nach Aufgabe 4



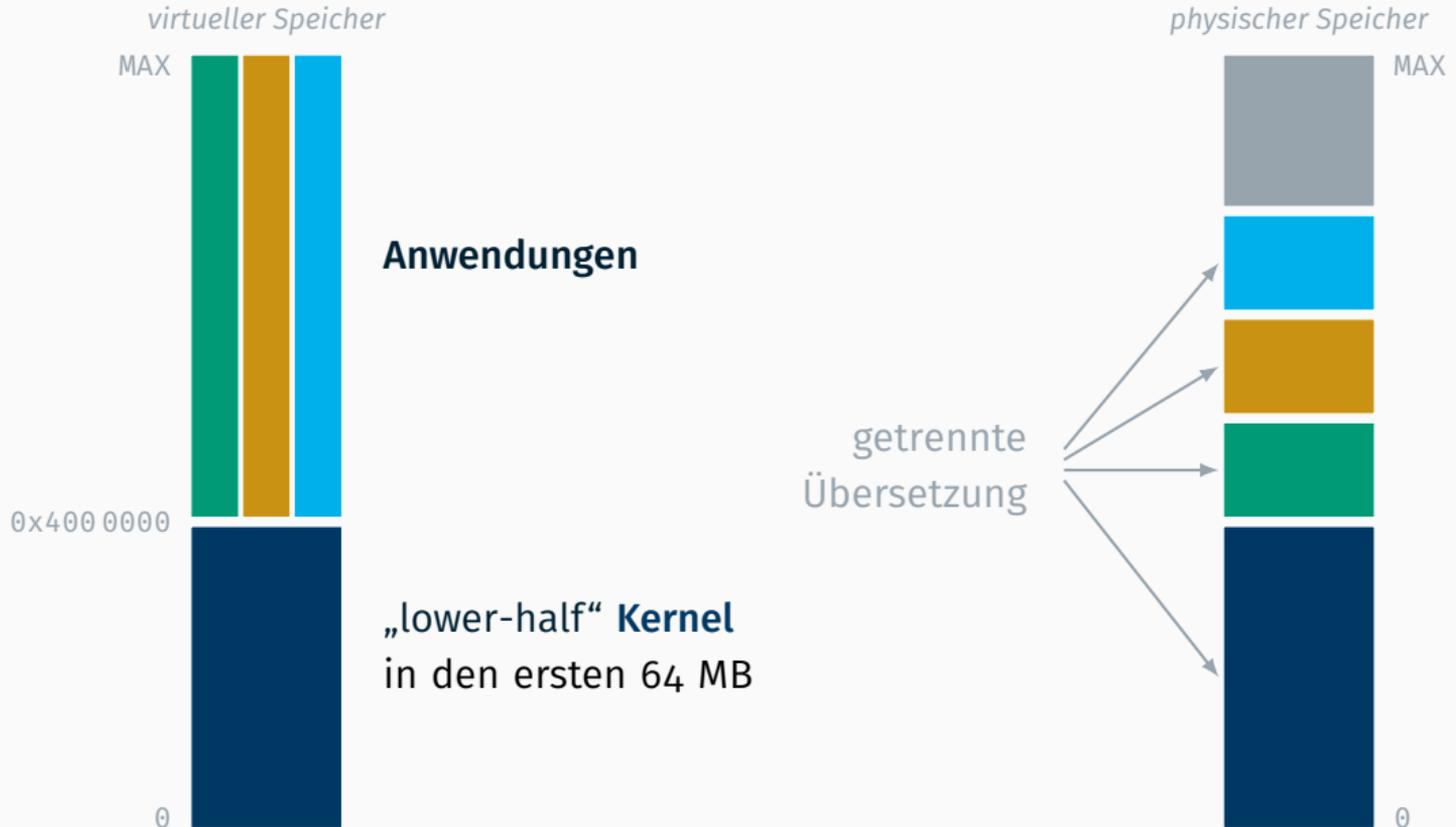
Anwendungen **1**, **2**, **x** auch untereinander  
getrennt

„lower-half“ **Kernel**  
in den ersten 64 MB

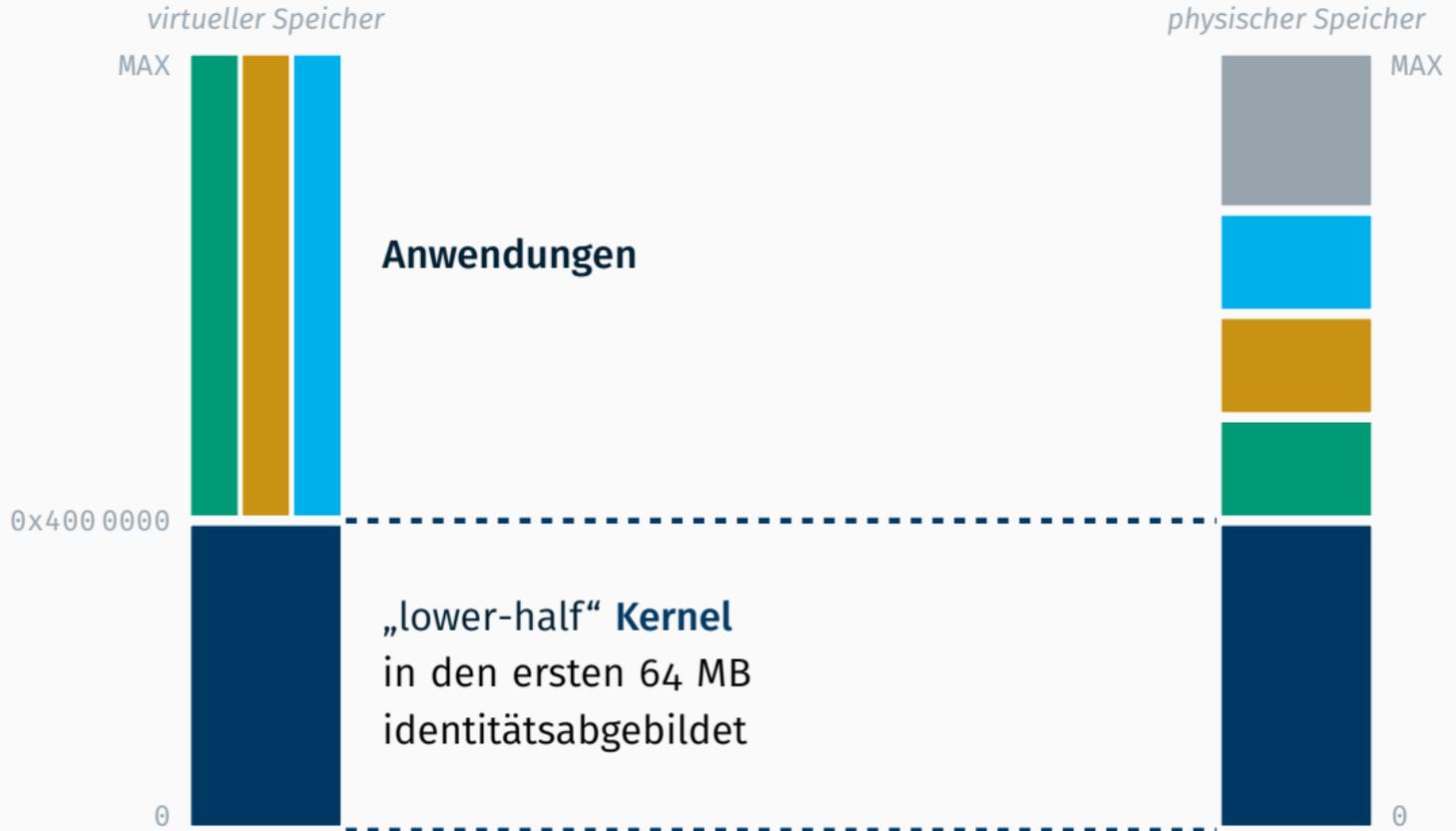
# Nach Aufgabe 4



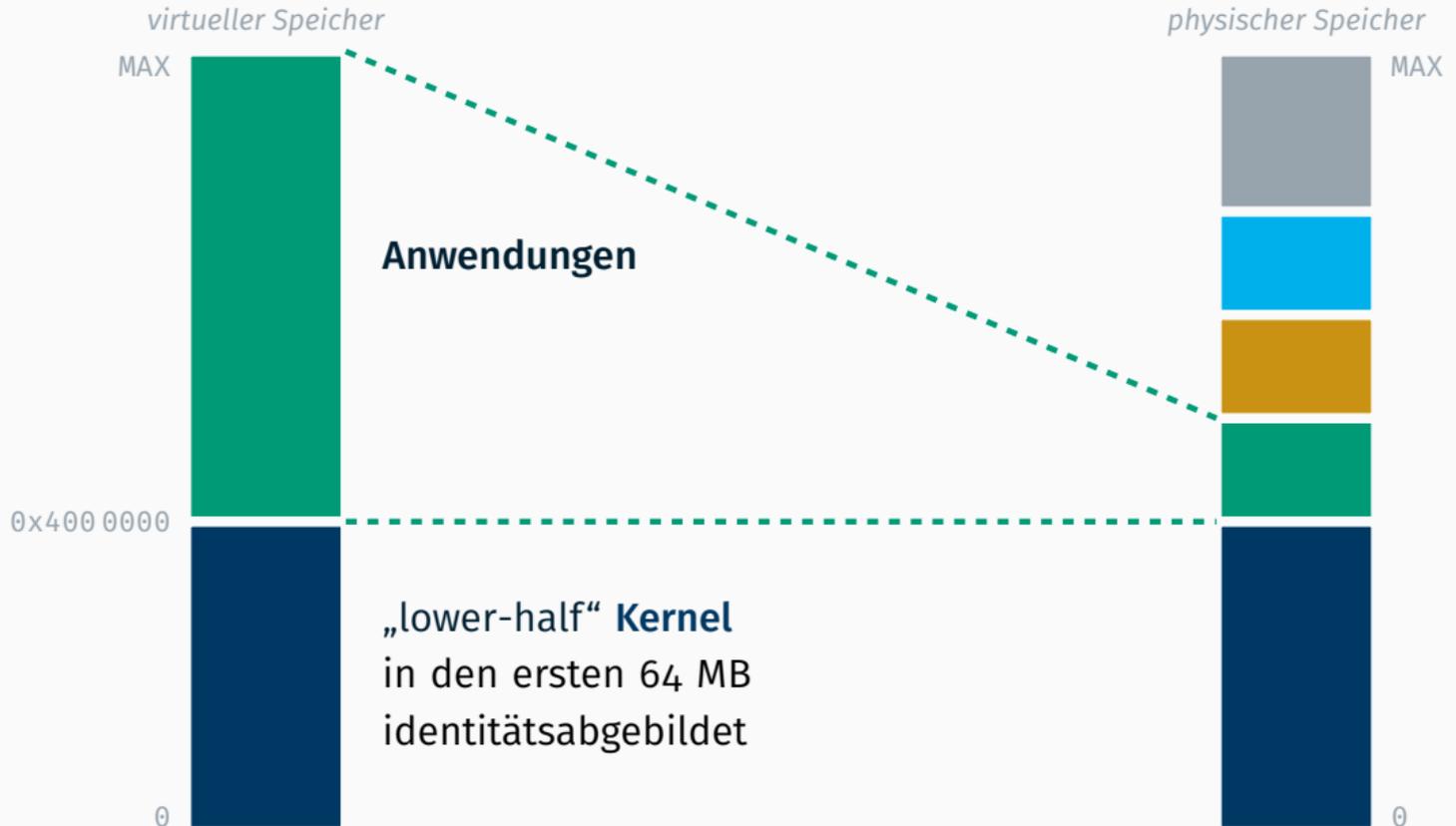
# Nach Aufgabe 4



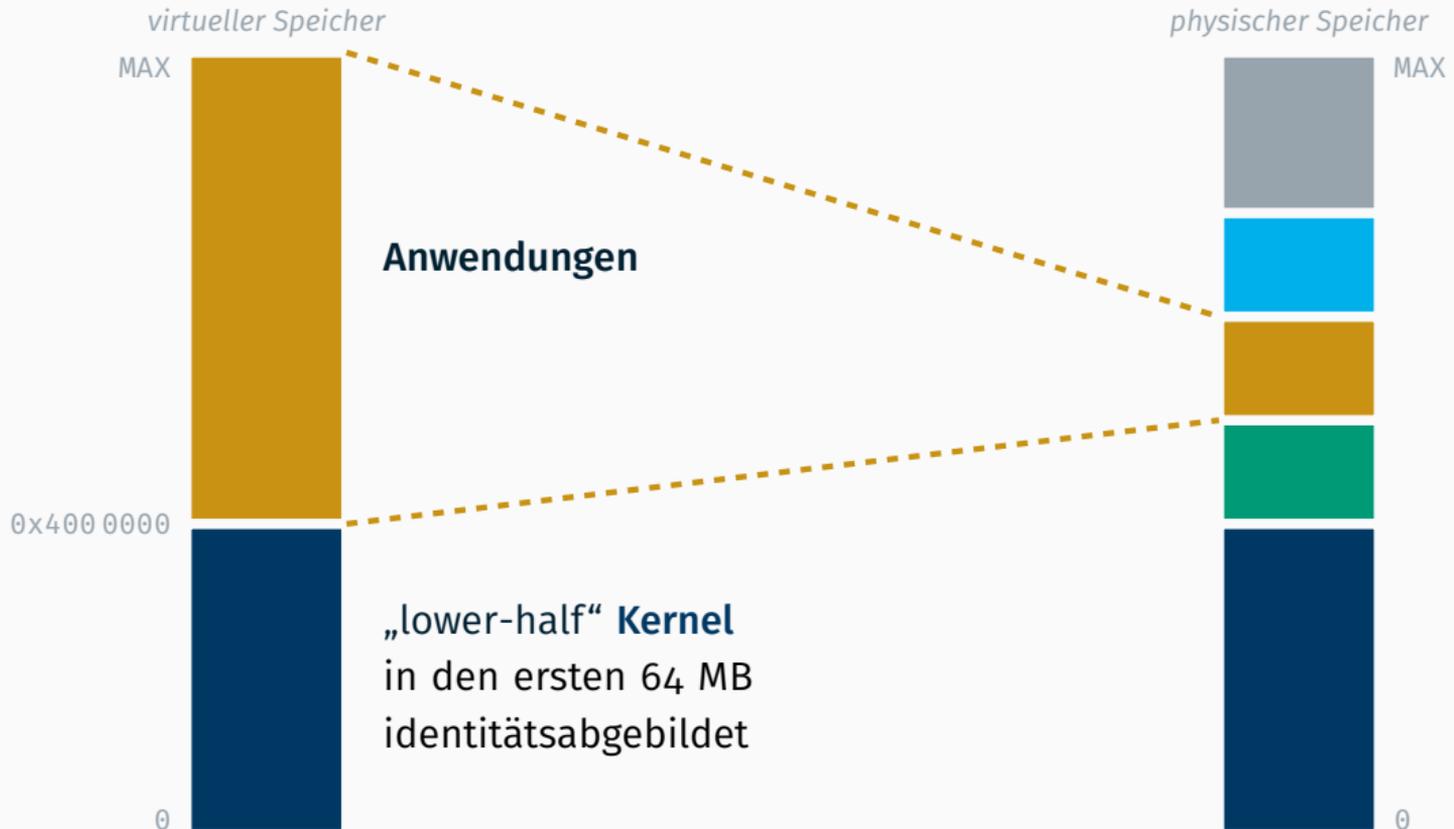
# Nach Aufgabe 4



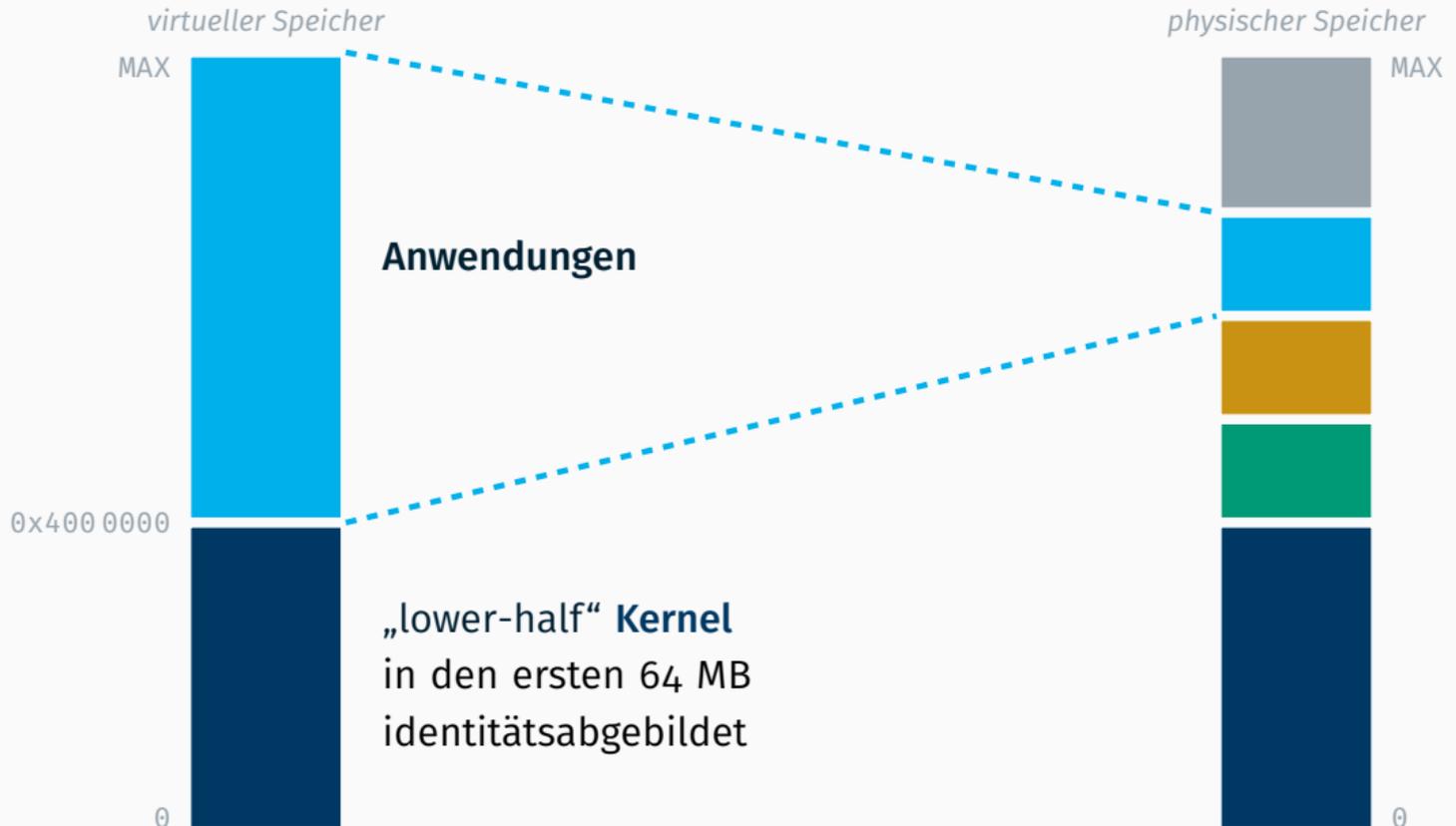
# Nach Aufgabe 4



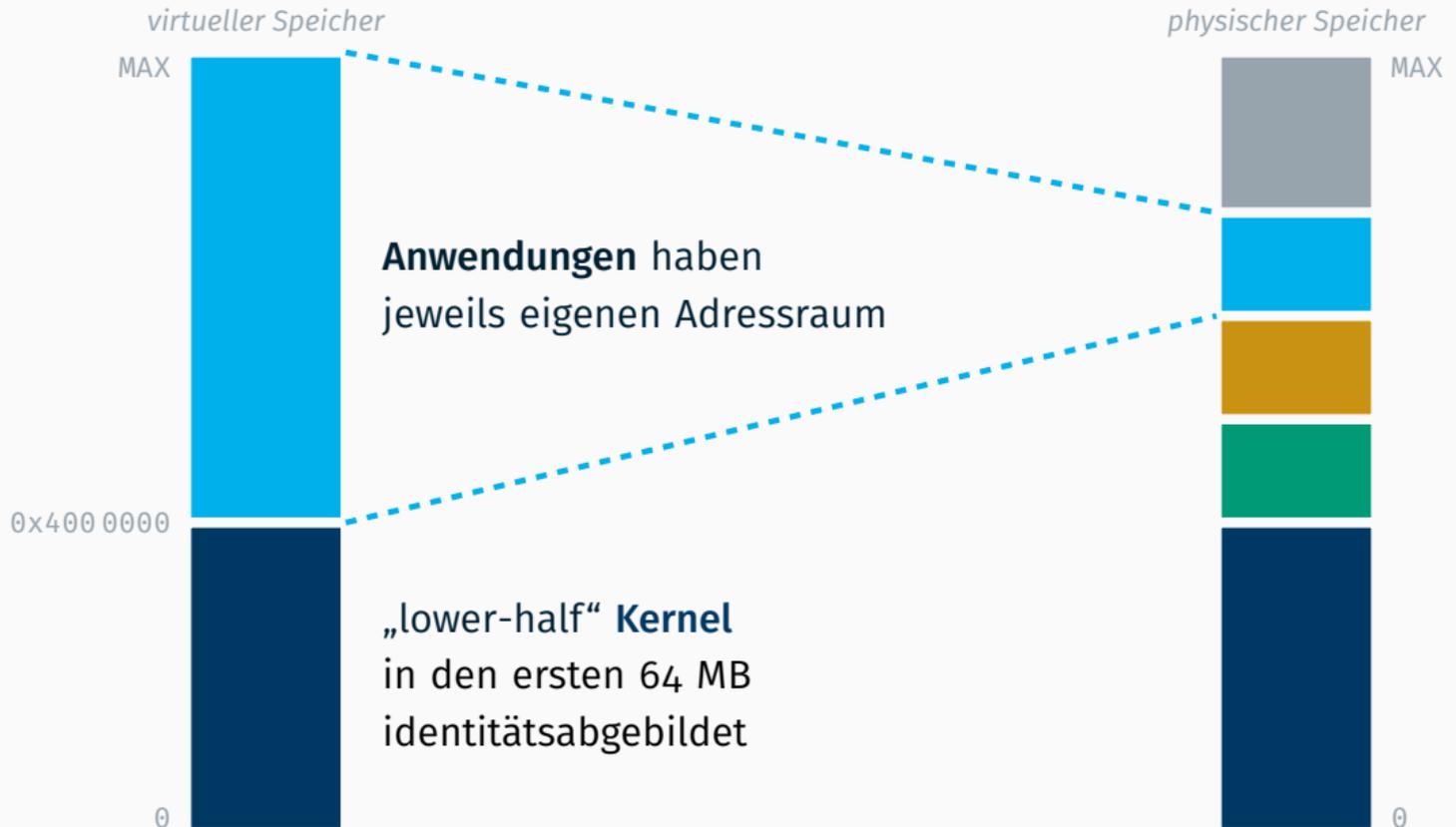
# Nach Aufgabe 4



# Nach Aufgabe 4



# Nach Aufgabe 4



***Exkurs:* MULTIBOOT SPECIFICATION**  
*oder* **Wie stiefel ich meinen Kernel?**

---

# Überblick zur MULTIBOOT SPECIFICATION

- offener PC **Bootloader Standard**, ab 1995 entwickelt

# Überblick zur MULTIBOOT SPECIFICATION

- offener PC **Bootloader Standard**, ab 1995 entwickelt
- Betriebssystem muss als **32 bit ELF** oder **a.out** vorliegen

# Überblick zur MULTIBOOT SPECIFICATION

- offener PC **Bootloader Standard**, ab 1995 entwickelt
- Betriebssystem muss als **32 bit ELF** oder **a.out** vorliegen
- übernimmt die [hässliche] Initialisierung eines x86 PCs in einen **wohl definierten Zustand**
  - 32 bit Protected Mode
  - nur BSP (*Bootstrap Processor*)
  - A20 Gate aktiviert
  - setzt optional auch Grafikmodus

# Überblick zur MULTIBOOT SPECIFICATION

- offener PC **Bootloader Standard**, ab 1995 entwickelt
- Betriebssystem muss als **32 bit ELF** oder **a.out** vorliegen
- übernimmt die [hässliche] Initialisierung eines x86 PCs in einen **wohl definierten Zustand**
  - 32 bit Protected Mode
  - nur BSP (*Bootstrap Processor*)
  - A20 Gate aktiviert
  - setzt optional auch Grafikmodus
- übergibt dem BS „**vitale**“ **Informationen** über das System

# Überblick zur MULTIBOOT SPECIFICATION

- offener PC **Bootloader Standard**, ab 1995 entwickelt
- Betriebssystem muss als **32 bit ELF** oder **a.out** vorliegen
- übernimmt die [hässliche] Initialisierung eines x86 PCs in einen **wohl definierten Zustand**
  - 32 bit Protected Mode
  - nur BSP (*Bootstrap Processor*)
  - A20 Gate aktiviert
  - setzt optional auch Grafikmodus
- übergibt dem BS „**vitale**“ **Informationen** über das System
- lädt ggf. auch *Boot Module* (weitere Dateien wie die **initiale Ramdisk**) in den Speicher

# Überblick zur MULTIBOOT SPECIFICATION

- offener PC **Bootloader Standard**, ab 1995 entwickelt
- Betriebssystem muss als **32 bit ELF** oder **a.out** vorliegen
- übernimmt die [hässliche] Initialisierung eines x86 PCs in einen **wohl definierten Zustand**
  - 32 bit Protected Mode
  - nur BSP (*Bootstrap Processor*)
  - A20 Gate aktiviert
  - setzt optional auch Grafikmodus
- übergibt dem BS „**vitale**“ **Informationen** über das System
- lädt ggf. auch *Boot Module* (weitere Dateien wie die **initiale Ramdisk**) in den Speicher
- wird u.a. von **GRUB** (Referenzimplementierung) und **PXELINUX** (Netzwerkboot) unterstützt

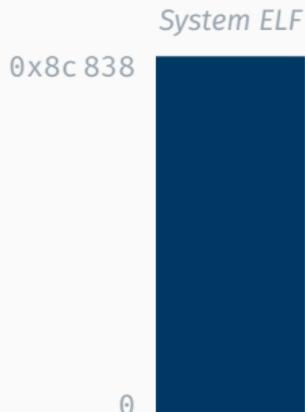
# Überblick zur MULTIBOOT SPECIFICATION

- offener PC **Bootloader Standard**, ab 1995 entwickelt
- Betriebssystem muss als **32 bit ELF** oder **a.out** vorliegen
- übernimmt die [hässliche] Initialisierung eines x86 PCs in einen **wohl definierten Zustand**
  - 32 bit Protected Mode
  - nur BSP (*Bootstrap Processor*)
  - A20 Gate aktiviert
  - setzt optional auch Grafikmodus
- übergibt dem BS „**vitale**“ **Informationen** über das System
- lädt ggf. auch *Boot Module* (weitere Dateien wie die **initiale Ramdisk**) in den Speicher
- wird u.a. von **GRUB** (Referenzimplementierung) und **PXELINUX** (Netzwerkboot) unterstützt
- und wird in **STUBS** verwendet

# Aufbau einer MULTIBOOT-kompatiblen Binärdatei

## Beispiel:

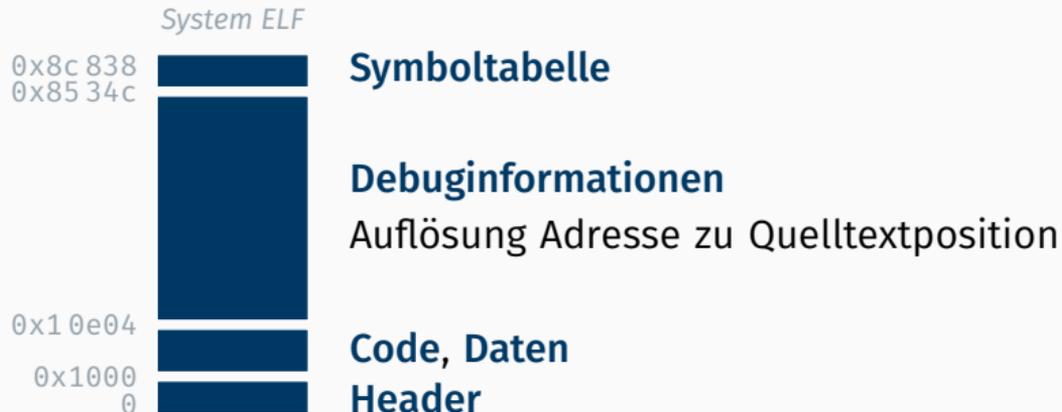
make generiert eine 563K große `.build/system` ELF.



# Aufbau einer MULTIBOOT-kompatiblen Binärdatei

## Beispiel:

make generiert eine 563K große `.build/system` ELF.  
Analyse mittels `readelf` offenbart folgende Struktur

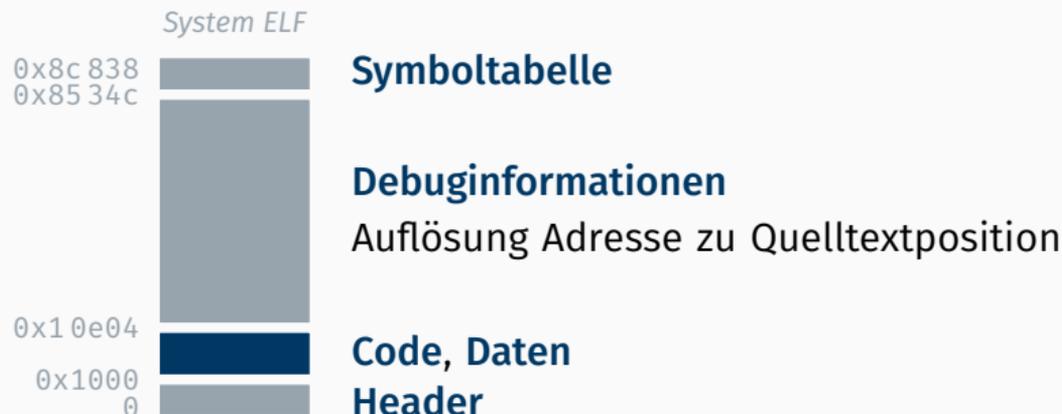


# Aufbau einer MULTIBOOT-kompatiblen Binärdatei

## Beispiel:

make generiert eine 563K große `.build/system` ELF.

Analyse mittels `readelf` offenbart folgende Struktur, für uns ist jedoch nur die (durch das Linkerskript) zusammengefasste Code- (`.text`) und Datensektion (`.[ro]data`) interessant

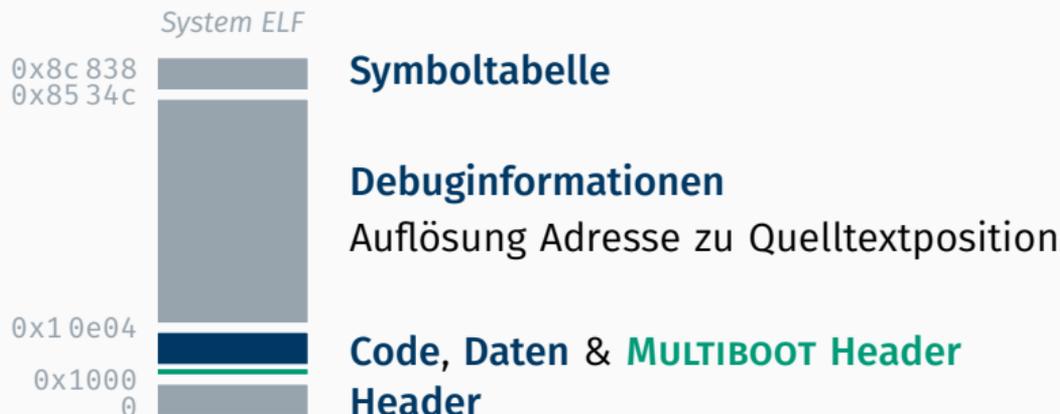


# Aufbau einer MULTIBOOT-kompatiblen Binärdatei

## Beispiel:

make generiert eine 563K große .build/system ELF.

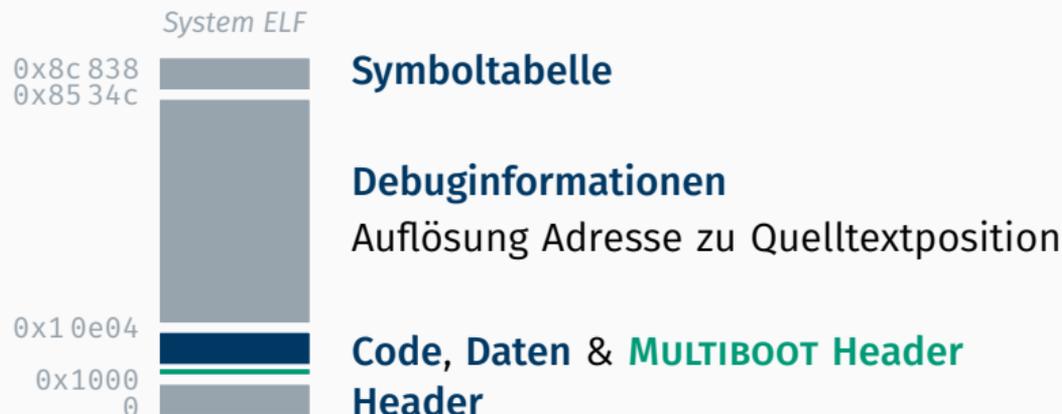
Analyse mittels readelf offenbart folgende Struktur, für uns ist jedoch nur die (durch das Linkerskript) zusammengefasste Code- (.text) und Datensektion (. [ro]data) interessant, in welcher auch der **MULTIBOOT Header** liegt



# Aufbau einer MULTIBOOT-kompatiblen Binärdatei

## MULTIBOOT Header

- Erkennung durch Wert 0x1bad b002 (und Prüfsumme)
- muss in den ersten 8192 Bytes (der ELF) liegen
- bei uns in boot/multiboot/header.asm definiert
- beinhaltet Konfiguration (via Flags)



```
[SECTION .multiboot_header]
; Constants included from boot/multiboot/config.inc
MULTIBOOT_HEADER_MAGIC_OS equ 0x1badb002 ; Magic Header

MULTIBOOT_PAGE_ALIGN      equ 1<<0      ; Align boot modules at 4K
MULTIBOOT_MEMORY_INFO     equ 1<<1      ; Request Memory Map info
MULTIBOOT_VIDEO_MODE      equ 1<<2      ; Configure video mode

MULTIBOOT_HEADER_FLAGS equ MULTIBOOT_PAGE_ALIGN | MULTIBOOT_MEMORY_INFO

MULTIBOOT_HEADER_CHKSUM equ -(MULTIBOOT_HEADER_MAGIC_OS + MULTIBOOT_HEADER_FLAGS)

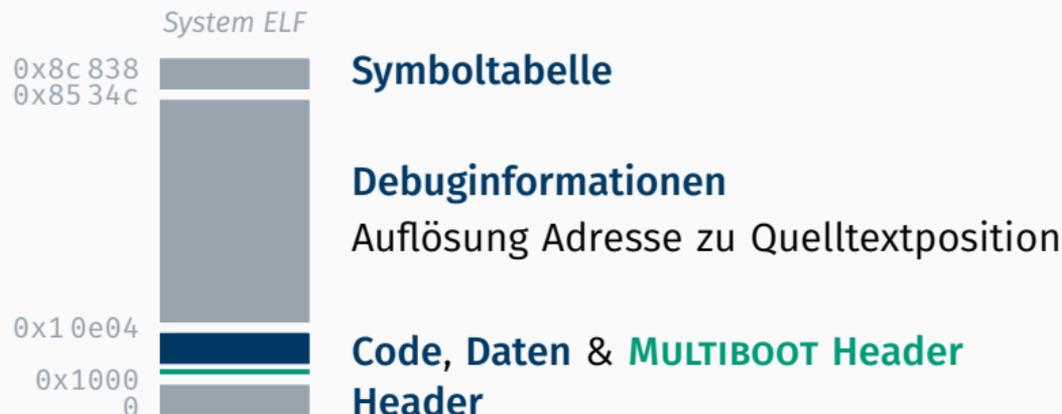
align 4                    ; Align section
multiboot_header:
    dd MULTIBOOT_HEADER_MAGIC_OS ; Magic Header Value
    dd MULTIBOOT_HEADER_FLAGS    ; Flags (affects following entries)
    dd MULTIBOOT_HEADER_CHKSUM   ; Header Checksum
    ; additional fields depending on flags
    ; (e.g. specifying the desired video mode)
```



# Aufbau einer MULTIBOOT-kompatiblen Binärdatei

## MULTIBOOT Header

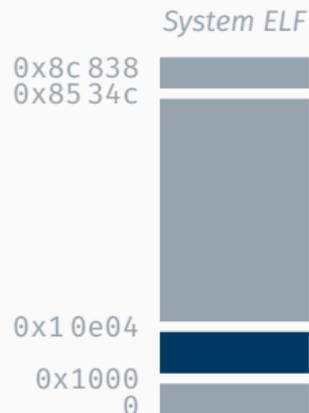
- Erkennung durch Wert 0x1bad b002 (und Prüfsumme)
- muss in den ersten 8192 Bytes (der ELF) liegen
- bei uns in boot/multiboot/header.asm definiert
- beinhaltet Konfiguration (via Flags)



# Laden einer MULTIBOOT-kompatiblen Binärdatei

## Ablauf im Bootloader

1. liest *System ELF*



*Hauptspeicher*

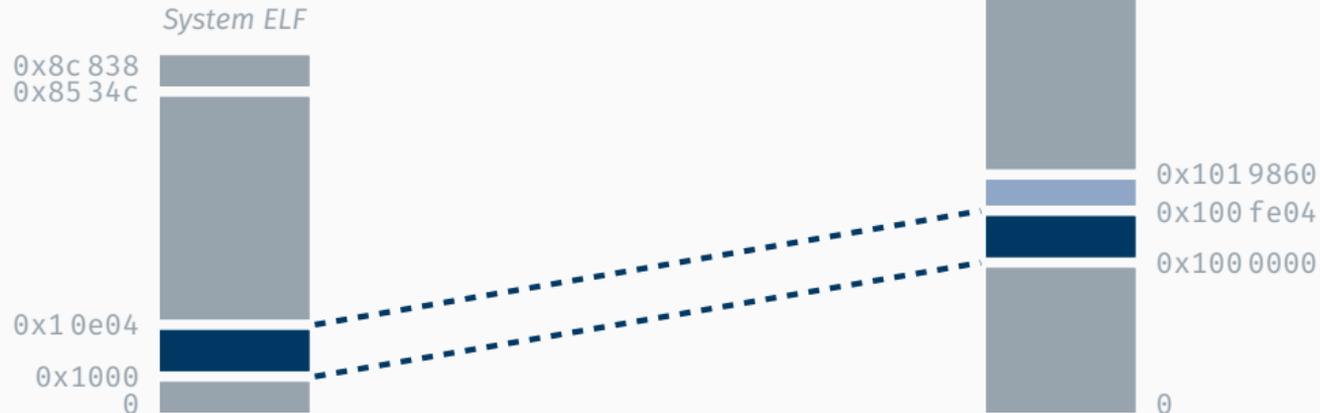


# Laden einer MULTIBOOT-kompatiblen Binärdatei

## Ablauf im Bootloader

1. liest *System ELF*
2. kopiert **Code & Datensektion** und erstellt **BSS**

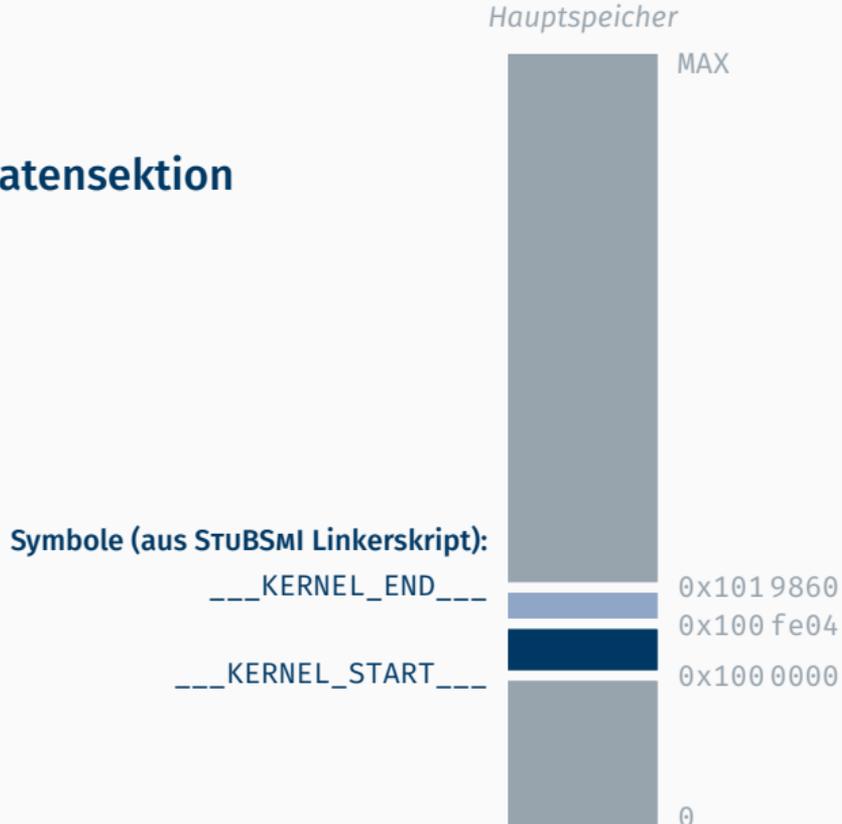
Hauptspeicher



# Laden einer MULTIBOOT-kompatiblen Binärdatei

## Ablauf im Bootloader

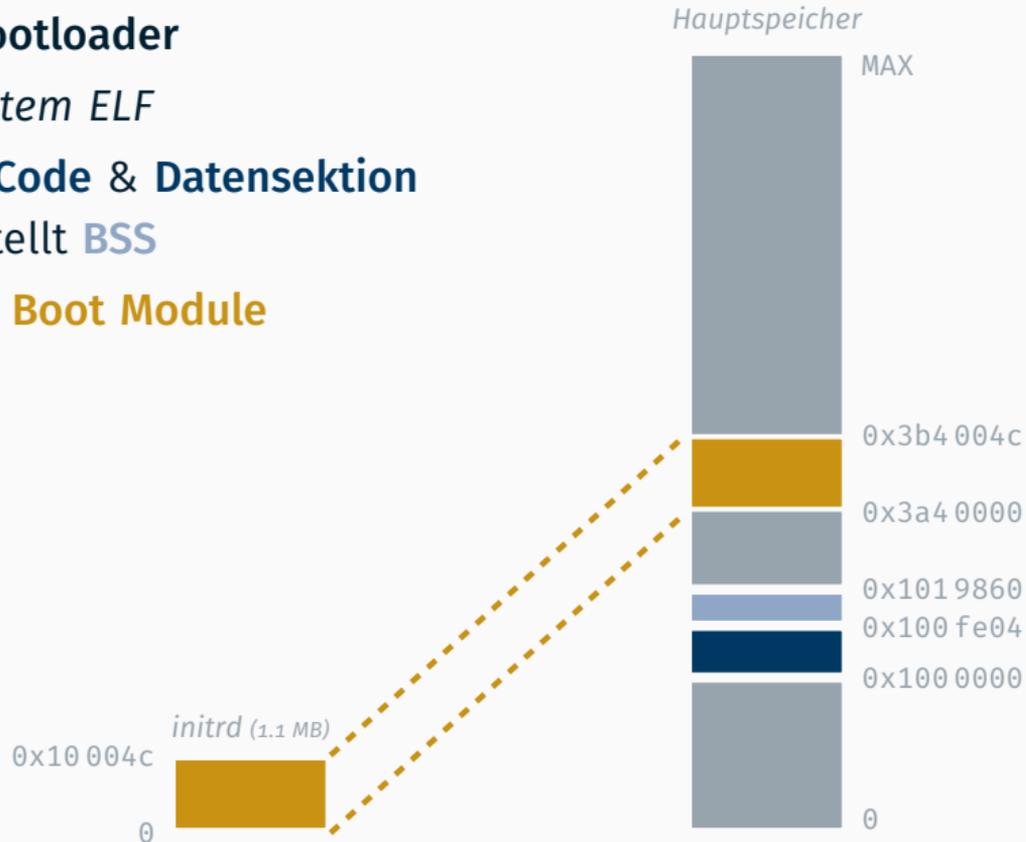
1. liest *System ELF*
2. kopiert **Code & Datensektion** und erstellt **BSS**



# Laden einer MULTIBOOT-kompatiblen Binärdatei

## Ablauf im Bootloader

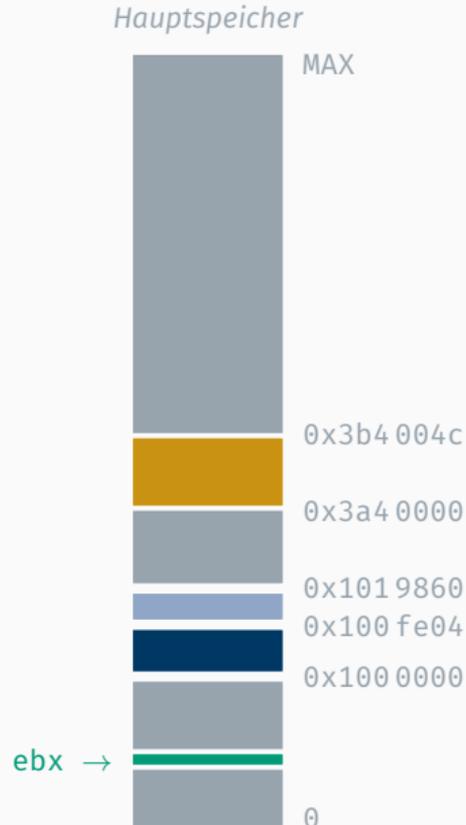
1. liest *System ELF*
2. kopiert **Code & Datensektion** und erstellt **BSS**
3. lädt ggf. **Boot Module**



# Laden einer MULTIBOOT-kompatiblen Binärdatei

## Ablauf im Bootloader

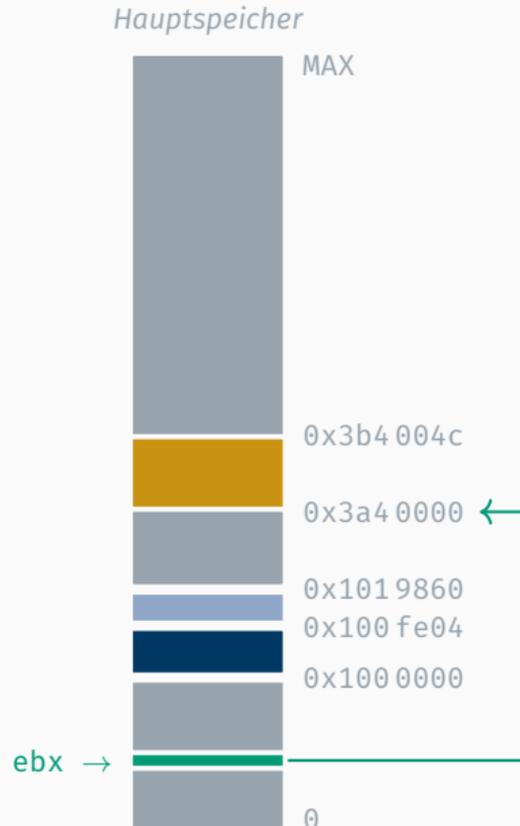
1. liest *System ELF*
2. kopiert **Code & Datensektion** und erstellt **BSS**
3. lädt ggf. **Boot Module**
4. setzt `eax` auf `0x2bad b002` sowie `ebx` als Zeiger auf Struktur mit **MULTIBOOT Information**



# Laden einer MULTIBOOT-kompatiblen Binärdatei

## Ablauf im Bootloader

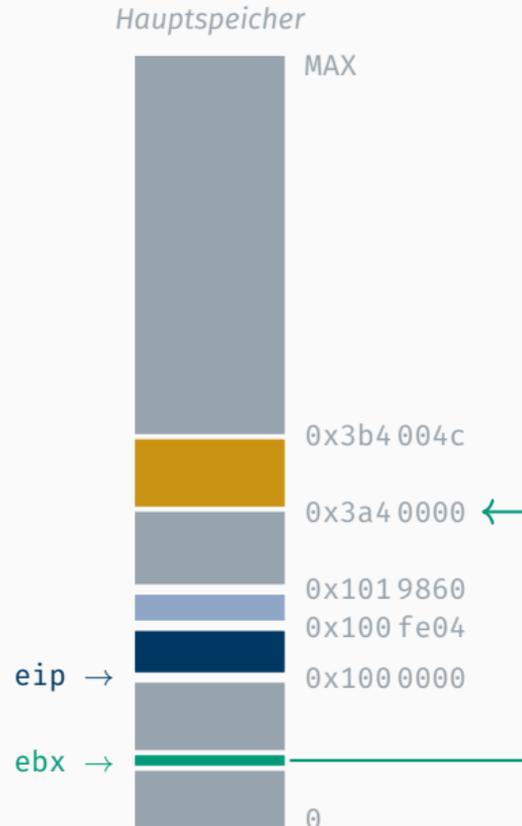
1. liest *System ELF*
2. kopiert **Code & Datensektion** und erstellt **BSS**
3. lädt ggf. **Boot Module**
4. setzt `eax` auf `0x2bad b002` sowie `ebx` als Zeiger auf Struktur mit **MULTIBOOT Information**



# Laden einer MULTIBOOT-kompatiblen Binärdatei

## Ablauf im Bootloader

1. liest *System ELF*
2. kopiert **Code & Datensektion** und erstellt **BSS**
3. lädt ggf. **Boot Module**
4. setzt `eax` auf `0x2bad b002` sowie `ebx` als Zeiger auf Struktur mit **MULTIBOOT Information**
5. Springt an den **Einsprungpunkt** (und übergibt somit an das Betriebssystem)



## Aufgabe 3

---

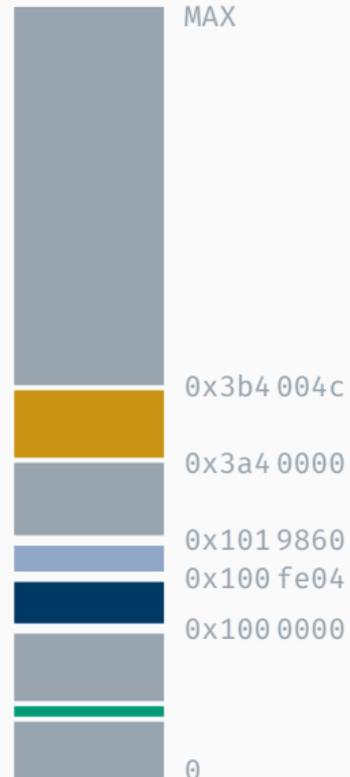
# Bestandsaufnahme

```
static Application apps[NUM_APPS];  
static KeyboardApplication kapp;
```

# Bestandsaufnahme

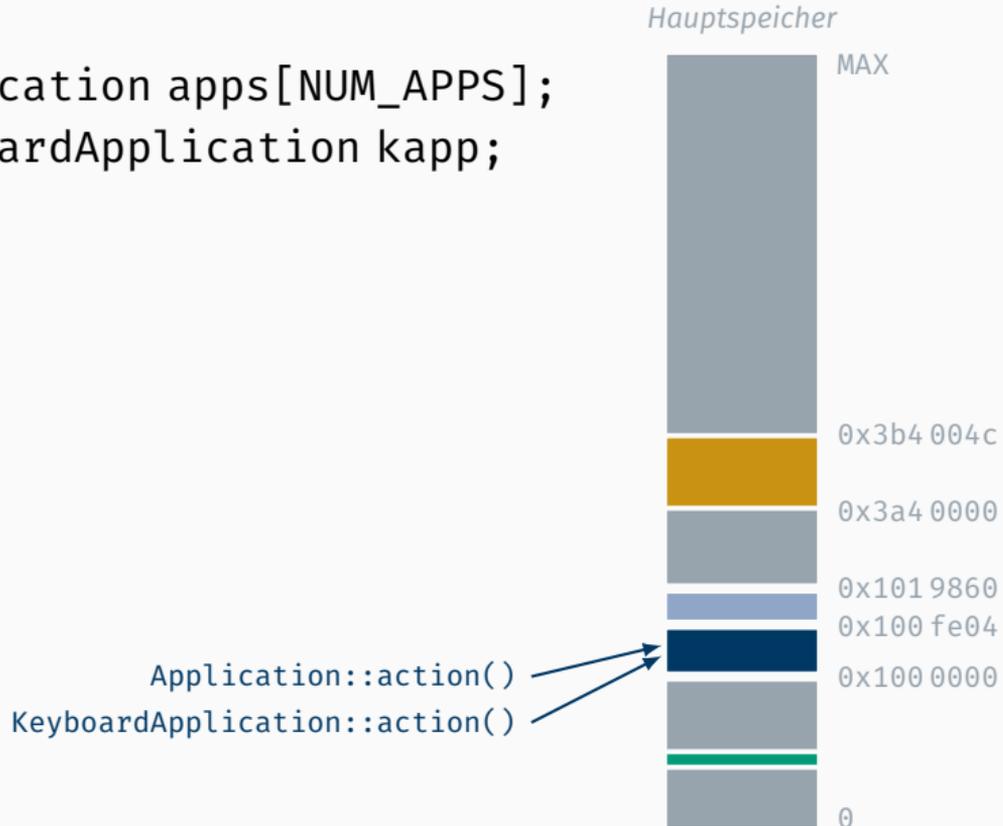
```
static Application apps[NUM_APPS];  
static KeyboardApplication kapp;
```

Hauptspeicher



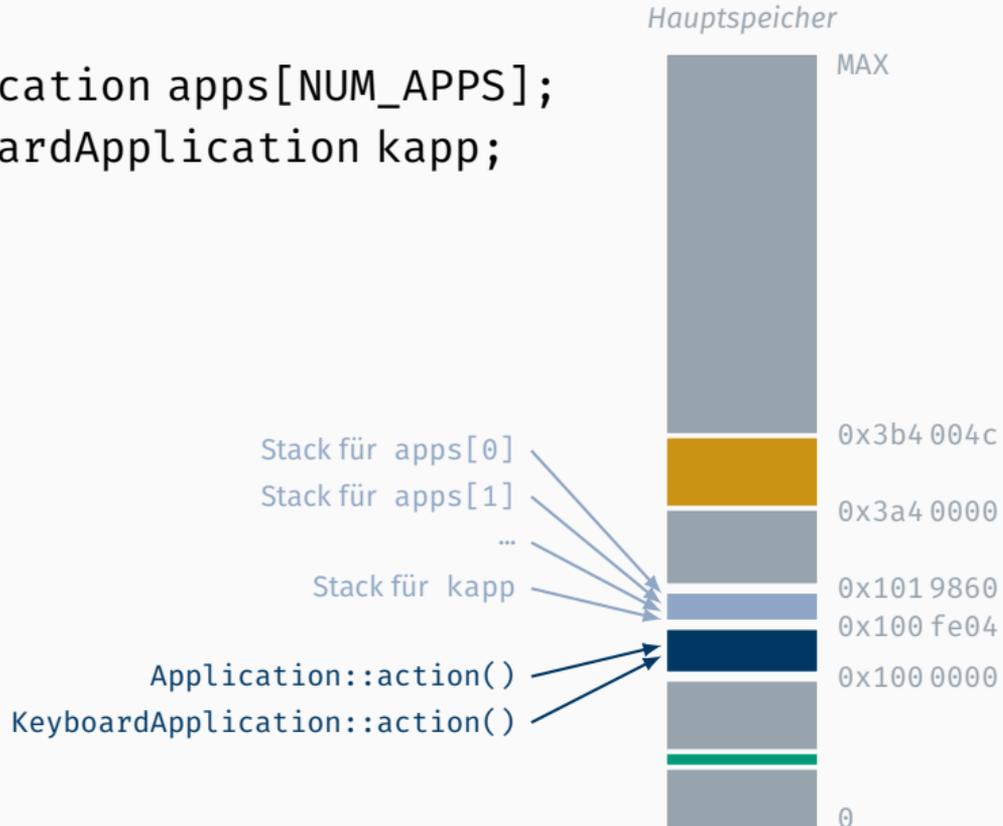
# Bestandsaufnahme

```
static Application apps[NUM_APPS];  
static KeyboardApplication kapp;
```



# Bestandsaufnahme

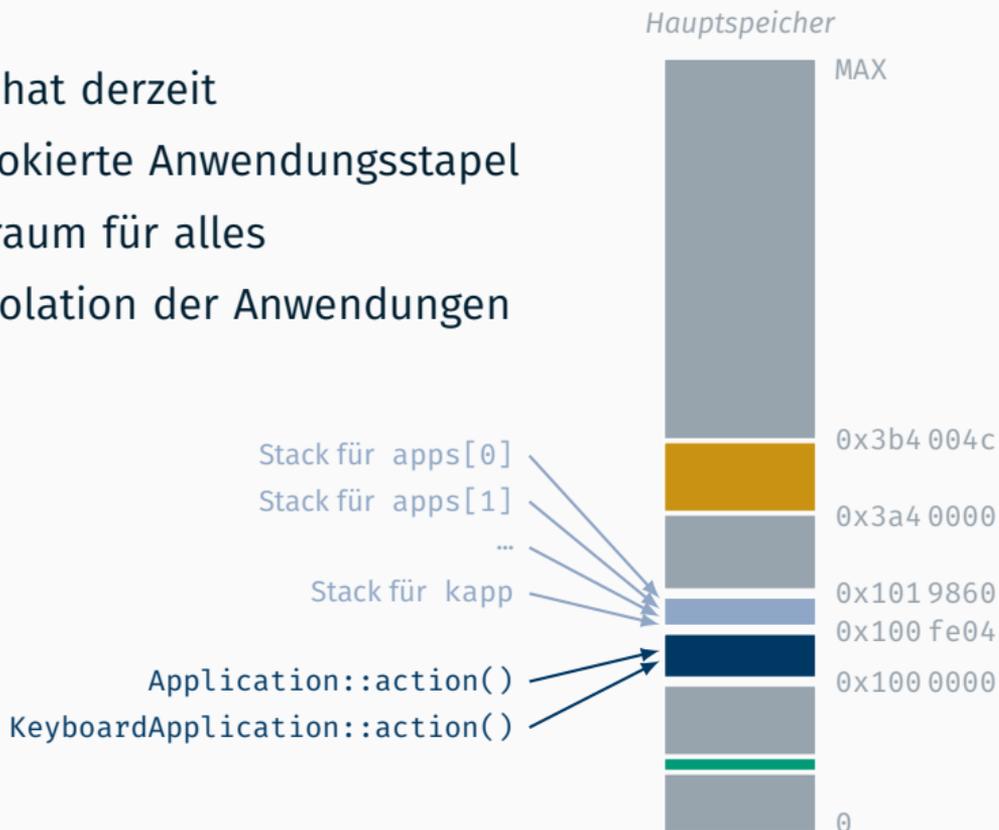
```
static Application apps[NUM_APPS];  
static KeyboardApplication kapp;
```



# Bestandsaufnahme

Unser **STUBSMI** hat derzeit

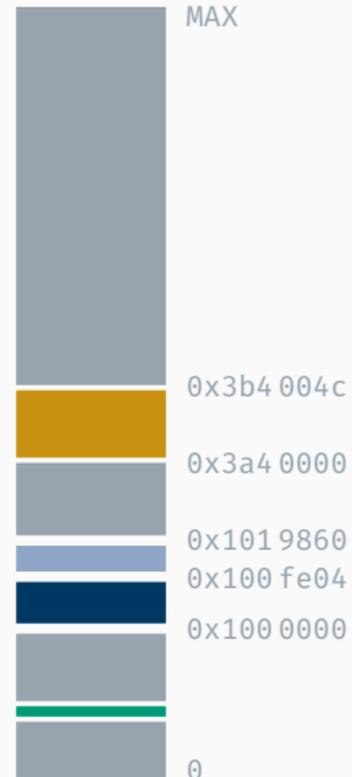
- statisch allokierte Anwendungsstapel
  - ein Adressraum für alles
- keinerlei Isolation der Anwendungen



# Aufgabeninhalt

**Ziel dieser Übung:** Anwendungen (ein wenig) voneinander isolieren

Hauptspeicher

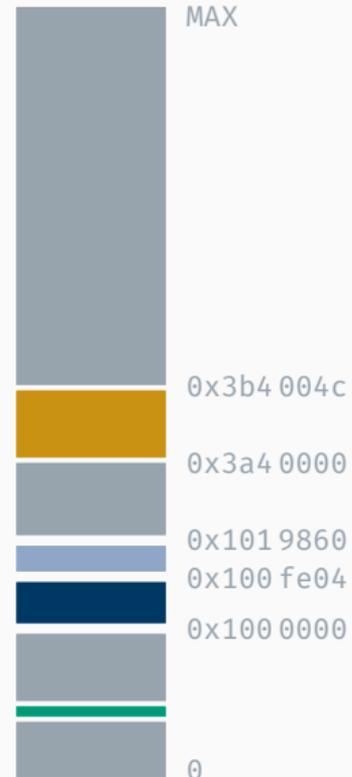


# Aufgabeninhalt

**Ziel dieser Übung:** Anwendungen (ein wenig) voneinander isolieren

- dynamisch Anwendungsstapel allokkieren

Hauptspeicher

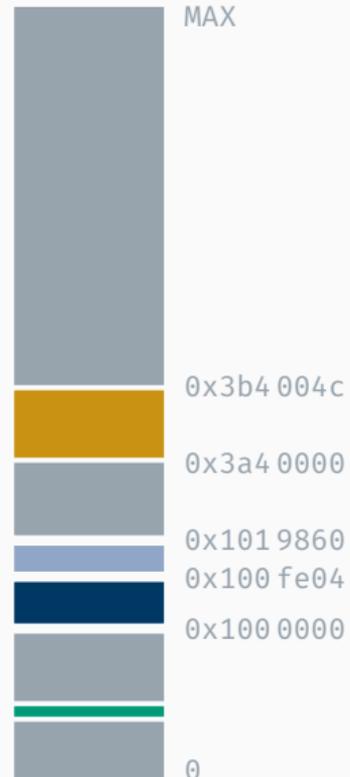


# Aufgabeninhalt

**Ziel dieser Übung:** Anwendungen (ein wenig) voneinander isolieren

- dynamisch Anwendungsstapel allokkieren
  - Hauptspeicherbelegung untersuchen

Hauptspeicher

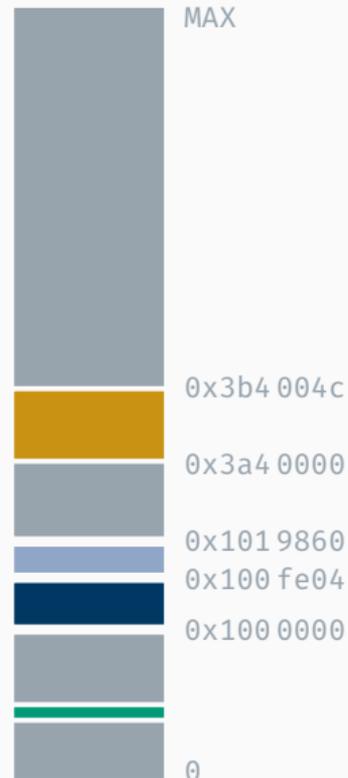


# Aufgabeninhalt

**Ziel dieser Übung:** Anwendungen (ein wenig) voneinander isolieren

- dynamisch Anwendungsstapel allokkieren
  - Hauptspeicherbelegung untersuchen
  - freien Speicher verwalten

Hauptspeicher

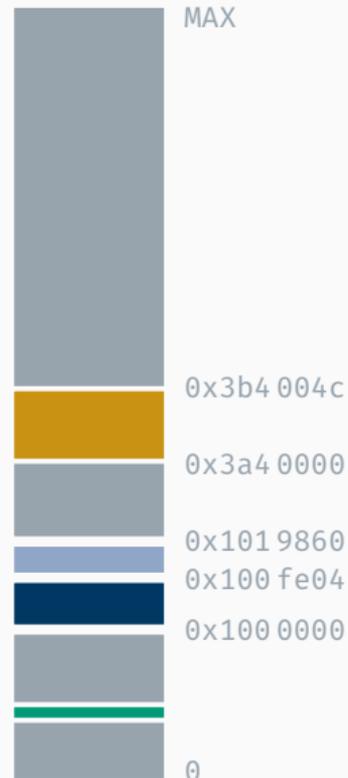


# Aufgabeninhalt

**Ziel dieser Übung:** Anwendungen (ein wenig) voneinander isolieren

- dynamisch Anwendungsstapel allokkieren
  - Hauptspeicherbelegung untersuchen
  - freien Speicher verwalten
- eigener Adressraum pro Anwendung

Hauptspeicher



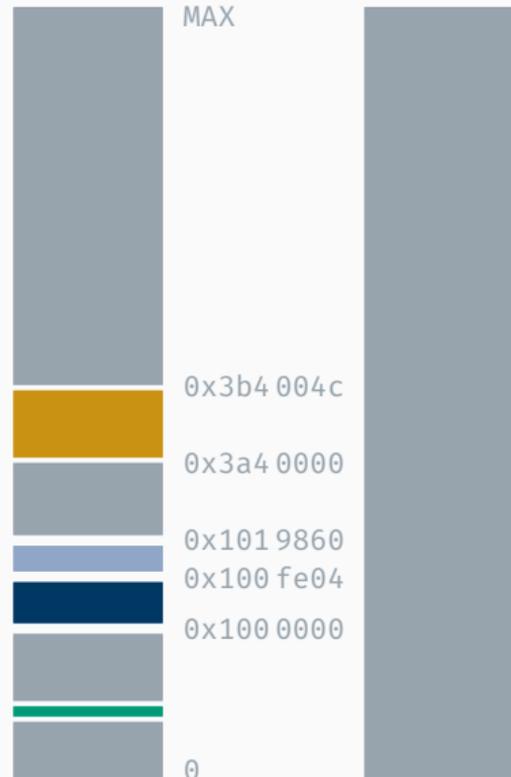
# Aufgabeninhalt

**Ziel dieser Übung:** Anwendungen (ein wenig) voneinander isolieren

- dynamisch Anwendungsstapel allokkieren
  - Hauptspeicherbelegung untersuchen
  - freien Speicher verwalten
- eigener Adressraum pro Anwendung
  - virtueller Adressraum mittels Paging

*physischer Speicher*

*virtueller Speicher*



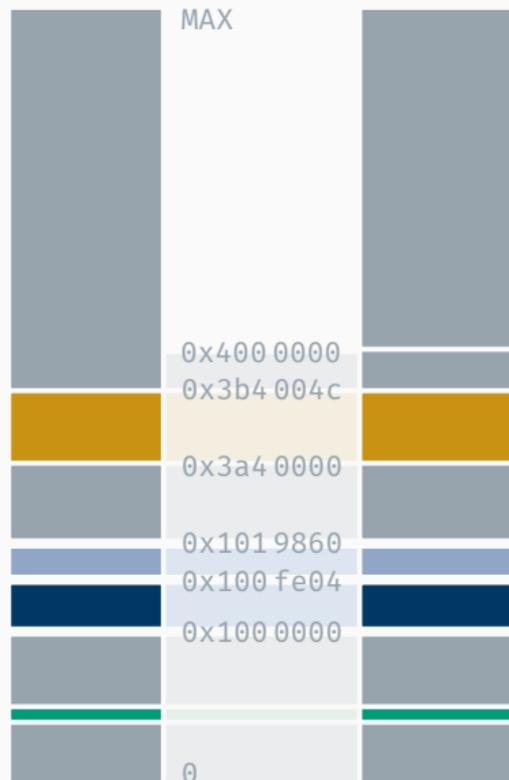
# Aufgabeninhalt

**Ziel dieser Übung:** Anwendungen (ein wenig) voneinander isolieren

- dynamisch Anwendungsstapel allokkieren
  - Hauptspeicherbelegung untersuchen
  - freien Speicher verwalten
- eigener Adressraum pro Anwendung
  - virtueller Adressraum mittels Paging
  - ersten 64 MB sind Kernelspace (lower-half) mit Identitätsabbildung

*physischer Speicher*

*virtueller Speicher*



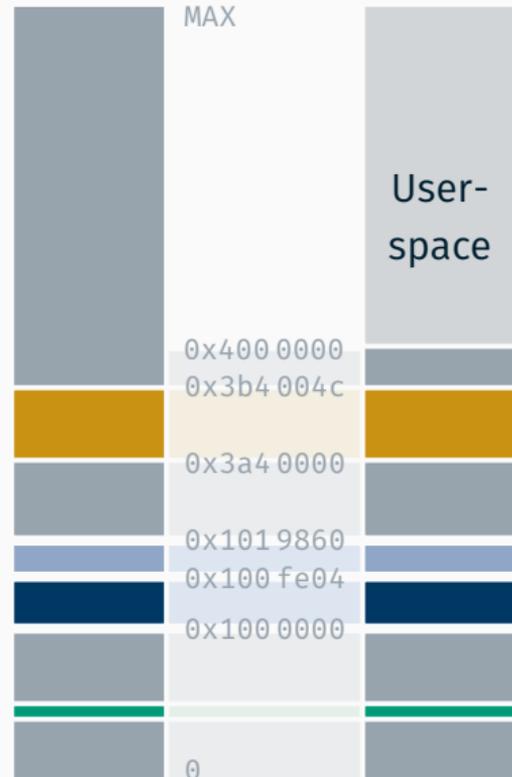
# Aufgabeninhalt

**Ziel dieser Übung:** Anwendungen (ein wenig) voneinander isolieren

- dynamisch Anwendungsstapel allokkieren
  - Hauptspeicherbelegung untersuchen
  - freien Speicher verwalten
- eigener Adressraum pro Anwendung
  - virtueller Adressraum mittels Paging
  - ersten 64 MB sind Kernelspace (lower-half) mit Identitätsabbildung
  - darüber liegt der Userspace

*physischer Speicher*

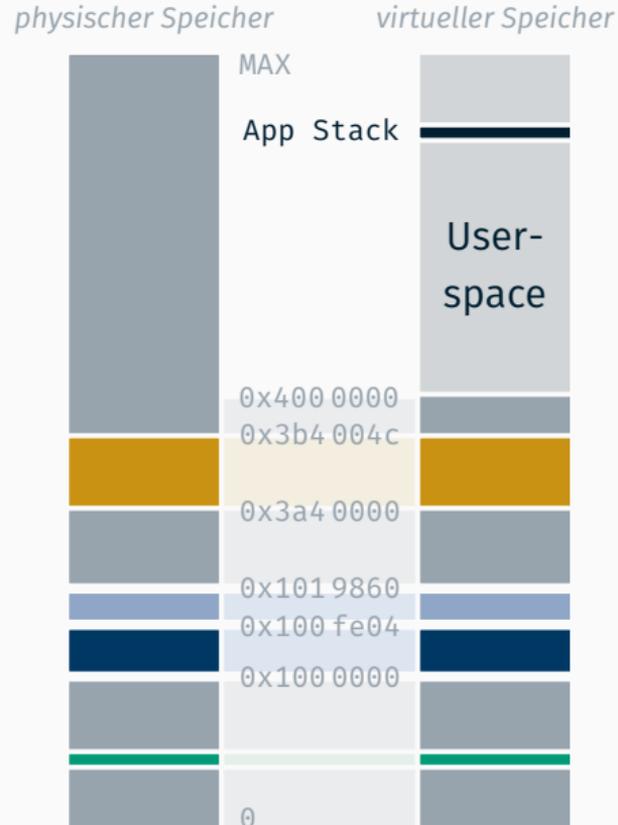
*virtueller Speicher*



# Aufgabeninhalt

**Ziel dieser Übung:** Anwendungen (ein wenig) voneinander isolieren

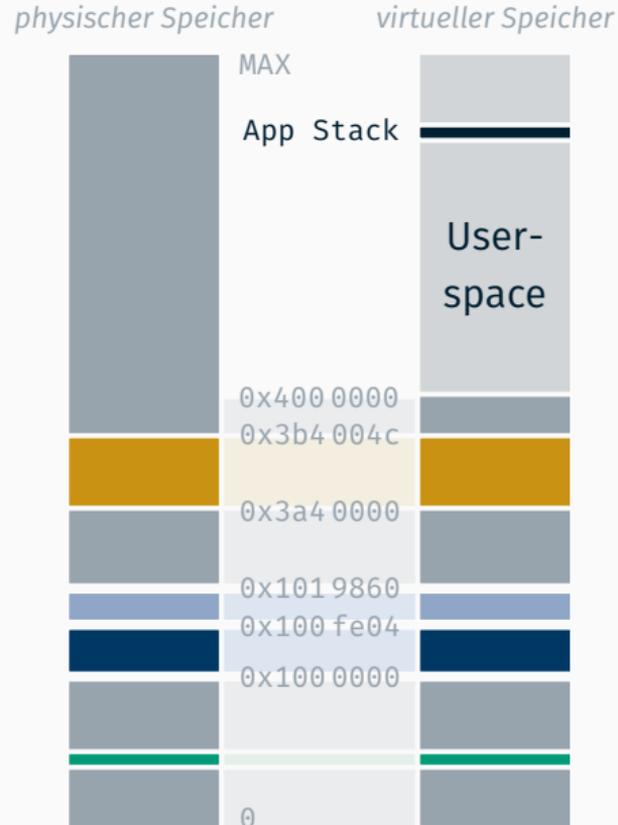
- dynamisch Anwendungsstapel allokkieren
  - Hauptspeicherbelegung untersuchen
  - freien Speicher verwalten
- eigener Adressraum pro Anwendung
  - virtueller Adressraum mittels Paging
  - ersten 64 MB sind Kernelspace (lower-half) mit Identitätsabbildung
  - darüber liegt der Userspace mit Anwendungsstapel



# Aufgabeninhalt

**Ziel dieser Übung:** Anwendungen (ein wenig) voneinander isolieren

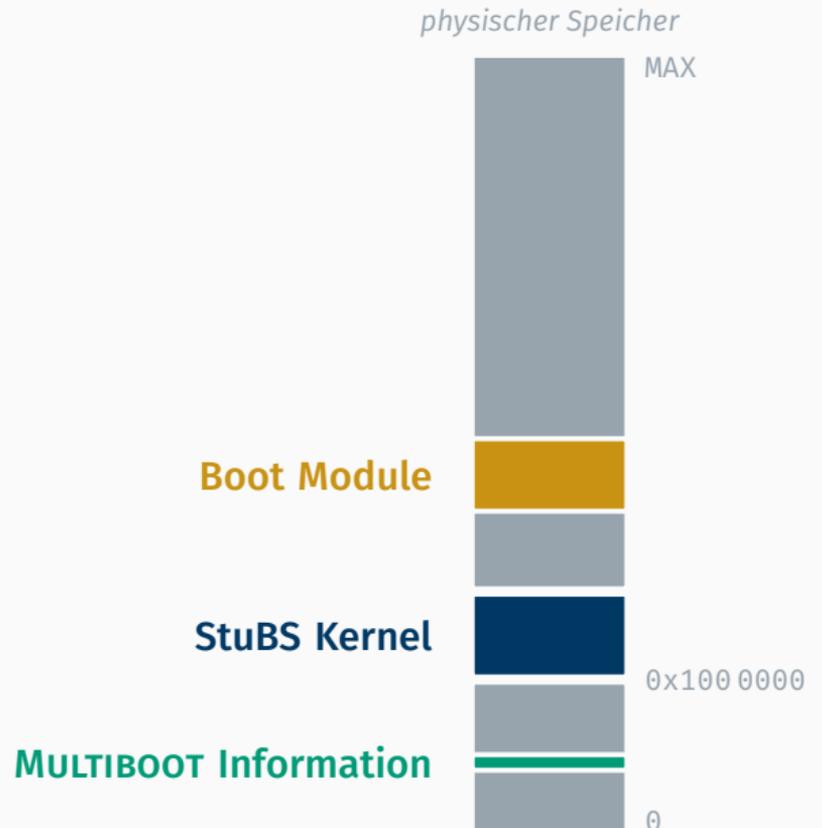
- dynamisch Anwendungsstapel allokalieren
  - Hauptspeicherbelegung untersuchen
  - freien Speicher verwalten
- eigener Adressraum pro Anwendung
  - virtueller Adressraum mittels Paging
  - ersten 64 MB sind Kernelspace (lower-half) mit Identitätsabbildung
  - darüber liegt der Userspace mit Anwendungsstapel an einer fixen Startadresse (z.B. Top-of-Stack bei 0x8000 0000 0000)



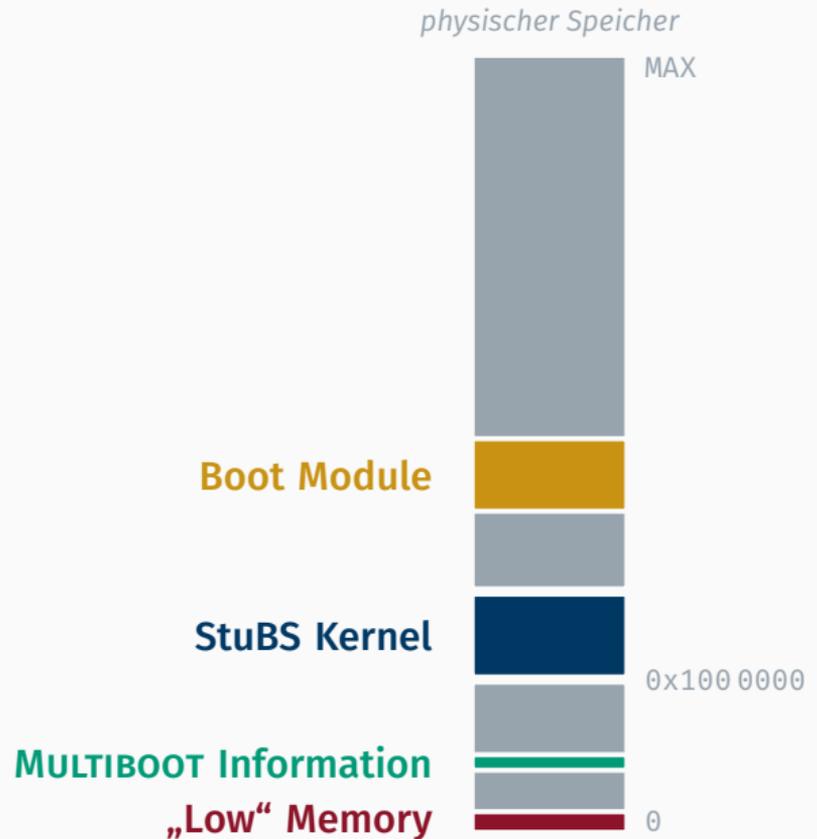
# Speicherverwaltung

---

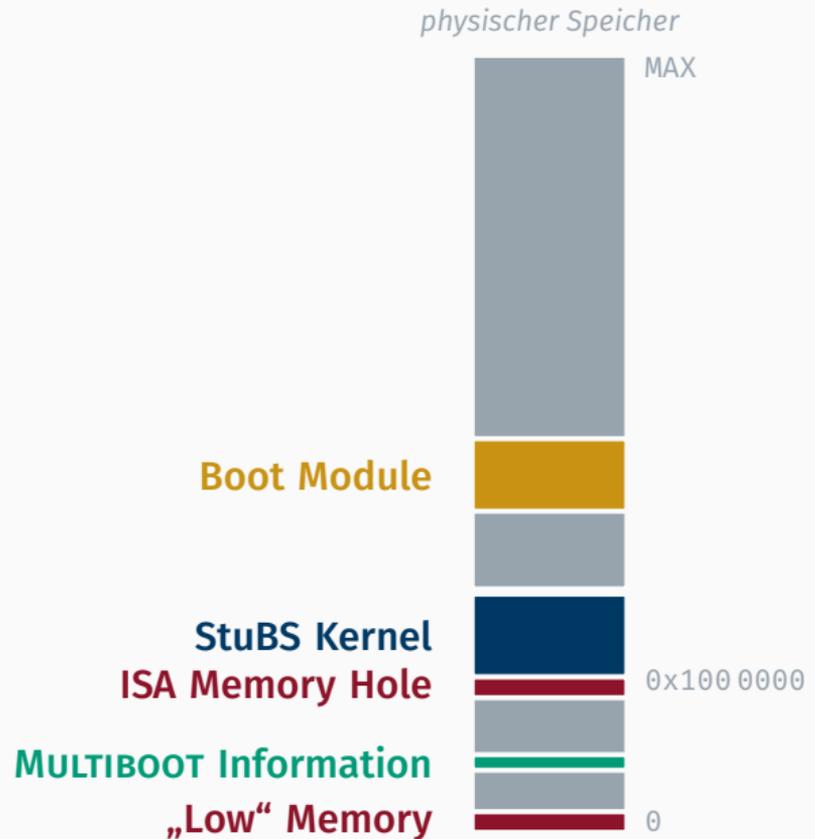
# Freien Speicher finden



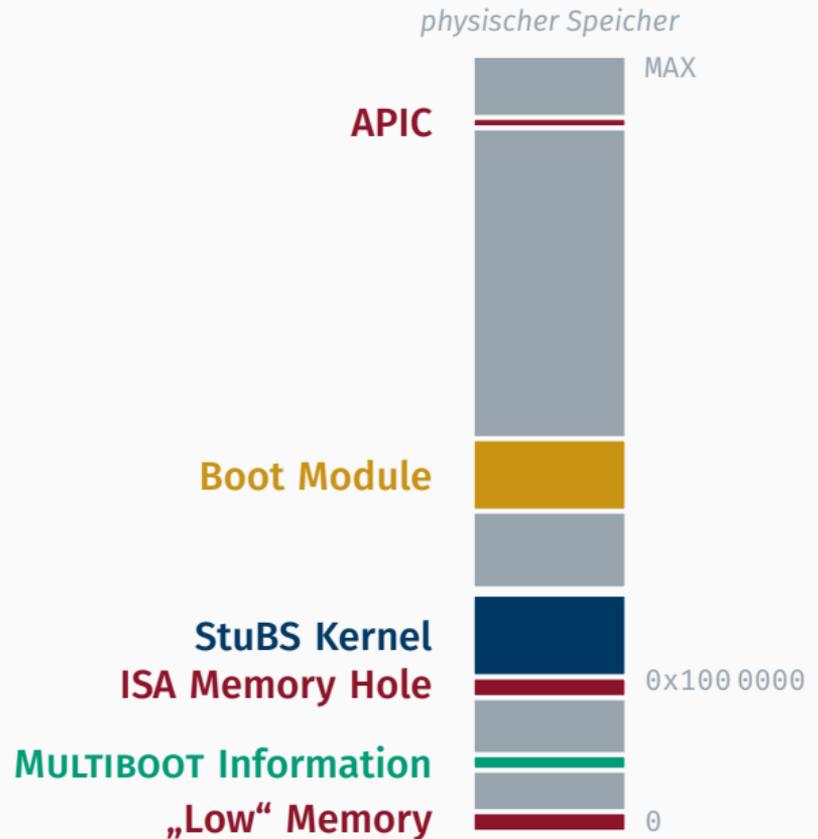
# Freien Speicher finden



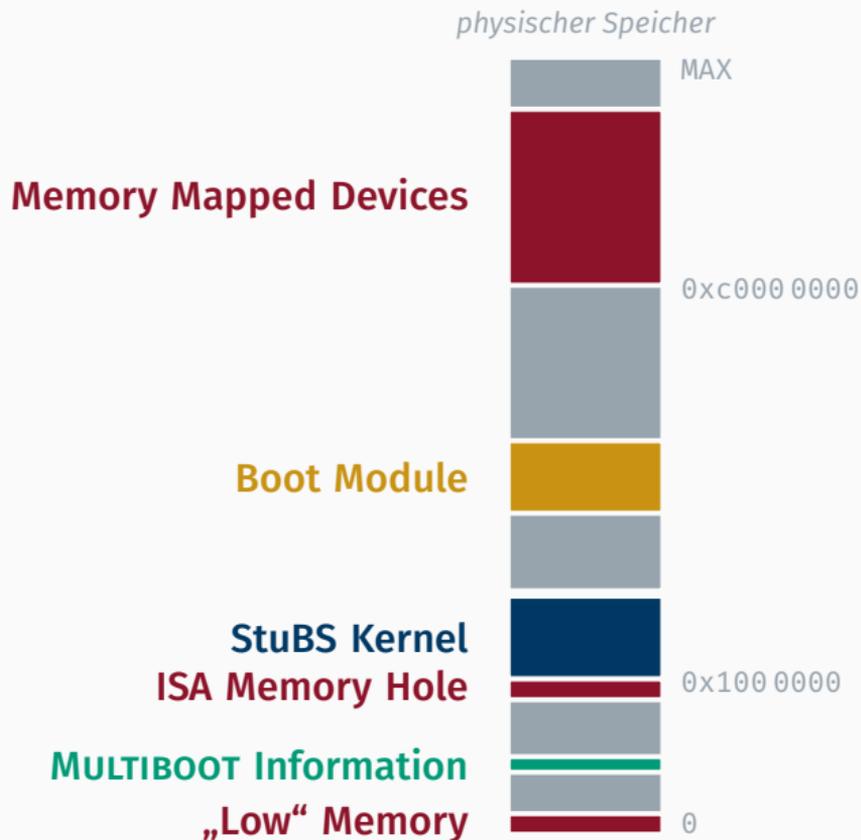
# Freien Speicher finden



# Freien Speicher finden



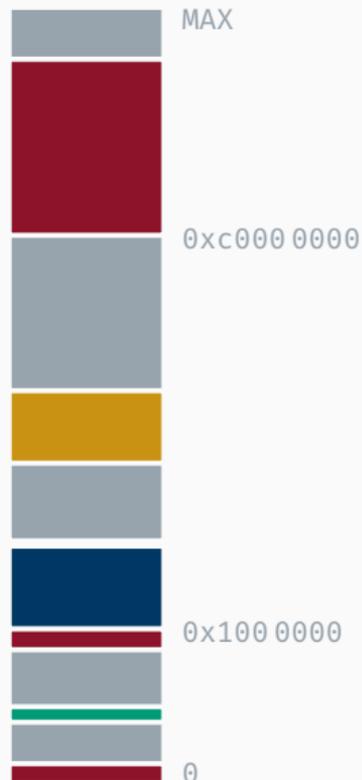
# Freien Speicher finden



# Freien Speicher finden

Abfrage der **Memory Map** über BIOS

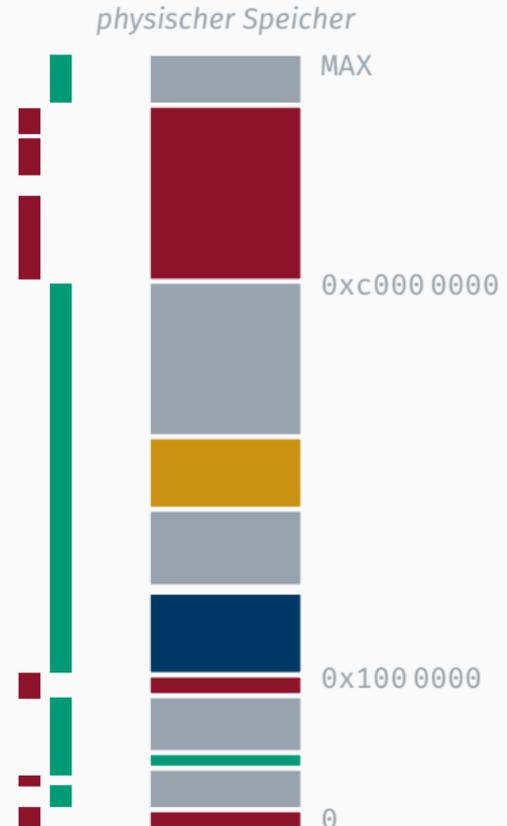
*physischer Speicher*



# Freien Speicher finden

Abfrage der **Memory Map** über BIOS  
Ergebnis in MULTIBOOT Information

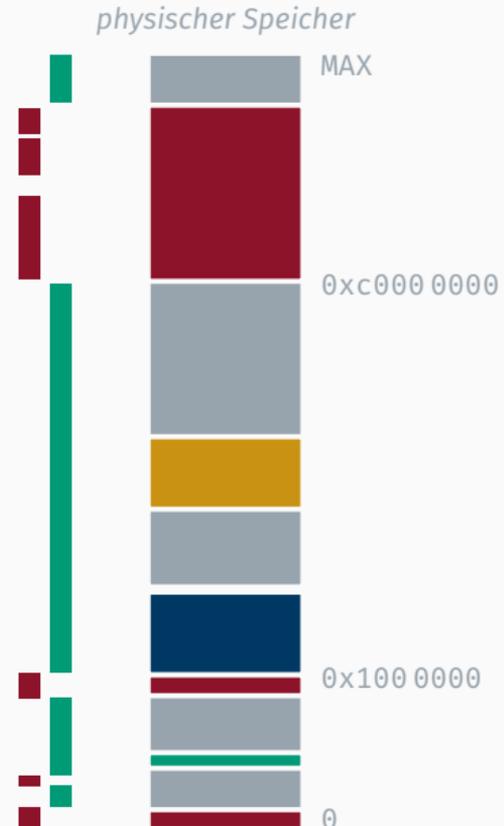
- freier und belegter Speicher



# Freien Speicher finden

Abfrage der **Memory Map** über BIOS  
Ergebnis in MULTIBOOT Information

- freier und belegter Speicher
- ignoriert aber später belegte Bereiche  
(wie Kernel, initrd und den MULTIBOOT Informationsblock)

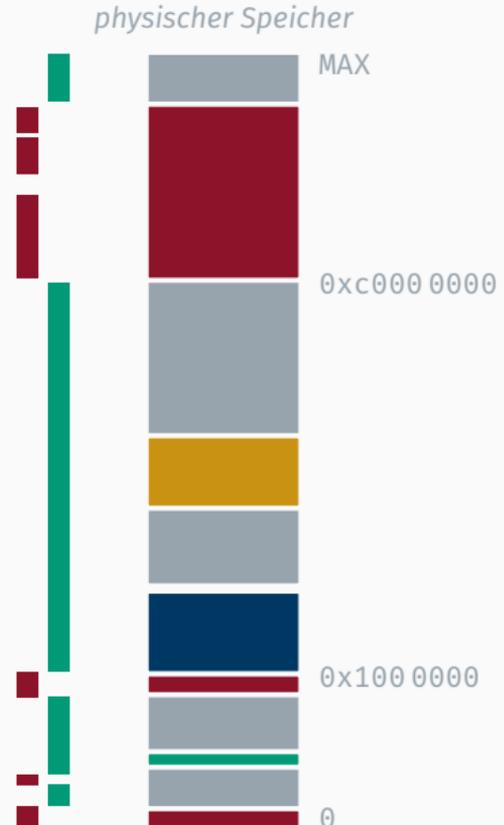


# Freien Speicher finden

Abfrage der **Memory Map** über BIOS

Ergebnis in MULTIBOOT Information

- freier und belegter Speicher
- ignoriert aber später belegte Bereiche  
(wie Kernel, initrd und den MULTIBOOT Informationsblock)
- besser defensiv auswerten  
(überlappende/widersprüchliche Bereiche möglich,  
ggf. fehlen im Speicher eingebündet Geräte [wie APIC])

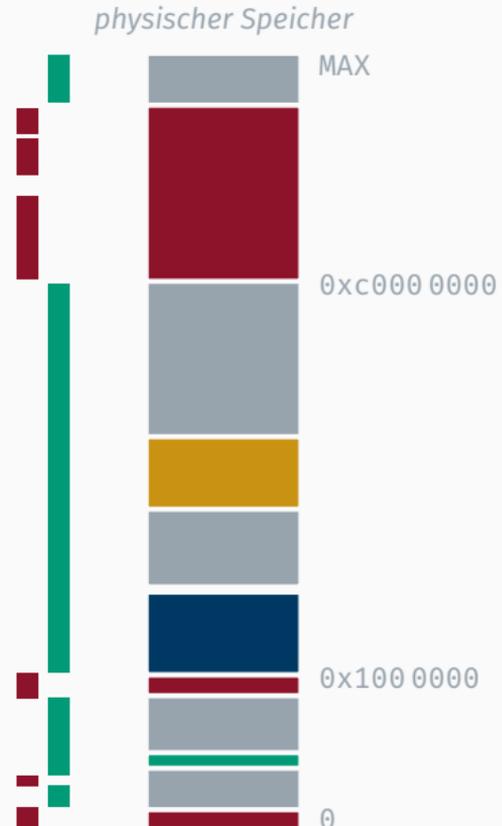


# Freien Speicher finden

Abfrage der **Memory Map** über BIOS

Ergebnis in MULTIBOOT Information

- freier und belegter Speicher
- ignoriert aber später belegte Bereiche  
(wie Kernel, initrd und den MULTIBOOT Informationsblock)
- besser defensiv auswerten  
(überlappende/widersprüchliche Bereiche möglich,  
ggf. fehlen im Speicher eingebündet Geräte [wie APIC])
- Verwaltung in geeigneter Struktur

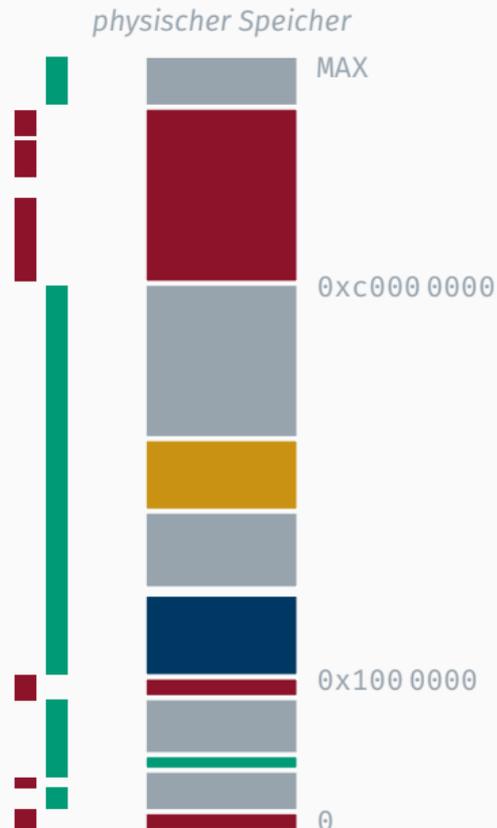


# Freien Speicher finden

Abfrage der **Memory Map** über BIOS

Ergebnis in MULTIBOOT Information

- **freier** und **belegter** Speicher
- ignoriert aber später belegte Bereiche  
(wie Kernel, initrd und den MULTIBOOT Informationsblock)
- besser defensiv auswerten  
(überlappende/widersprüchliche Bereiche möglich,  
ggf. fehlen im Speicher eingeblendet Geräte [wie APIC])
- Verwaltung in geeigneter Struktur
  - verkettete Liste mit freien Seitenbereichen  
(Startadresse, Länge) für uns ausreichend

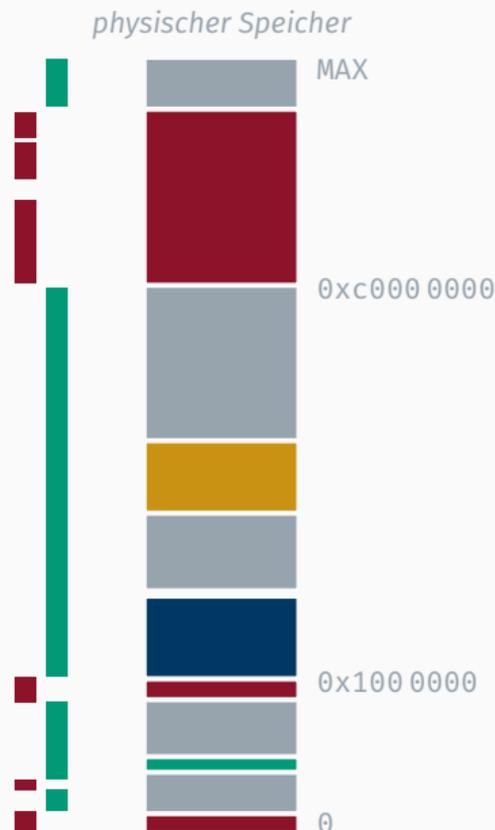


# Freien Speicher finden

Abfrage der **Memory Map** über BIOS

Ergebnis in MULTIBOOT Information

- freier und belegter Speicher
- ignoriert aber später belegte Bereiche  
(wie Kernel, initrd und den MULTIBOOT Informationsblock)
- besser defensiv auswerten  
(überlappende/widersprüchliche Bereiche möglich, ggf. fehlen im Speicher eingeblendet Geräte [wie APIC])
- Verwaltung in geeigneter Struktur
  - verkettete Liste mit freien Seitenbereichen (Startadresse, Länge) für uns ausreichend
  - dynamische Allokation der Listenelemente zulässig (→ `utils/alloc.h`)

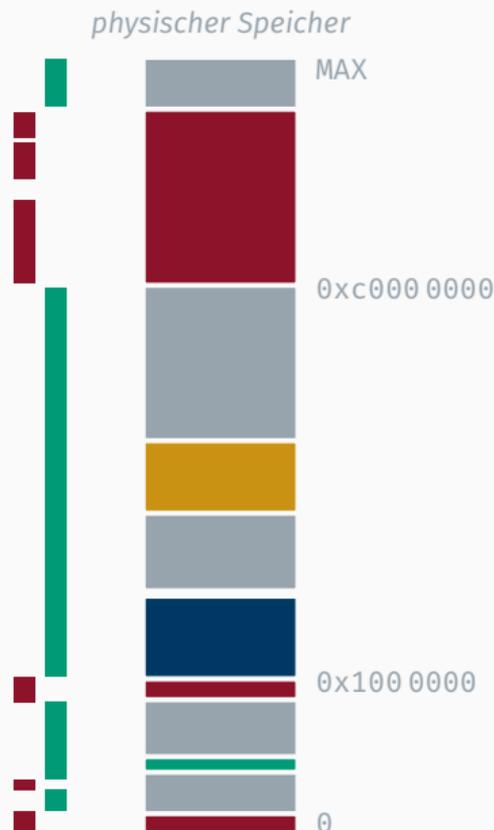


# Freien Speicher finden

Abfrage der **Memory Map** über BIOS

Ergebnis in MULTIBOOT Information

- freier und belegter Speicher
- ignoriert aber später belegte Bereiche  
(wie Kernel, initrd und den MULTIBOOT Informationsblock)
- besser defensiv auswerten  
(überlappende/widersprüchliche Bereiche möglich,  
ggf. fehlen im Speicher eingeblendet Geräte [wie APIC])
- Verwaltung in geeigneter Struktur
  - verkettete Liste mit freien Seitenbereichen  
(Startadresse, Länge) für uns ausreichend
  - dynamische Allokation der Listenelemente  
zulässig (→ `utils/alloc.h`)
  - Unterscheidung zw. Kernel- und Userspace





Der *Page Frame Allocator* sollte unbedingt vor dem nachfolgenden Schritt ausgiebig getestet werden!



Der *Page Frame Allocator* sollte unbedingt vor dem nachfolgenden Schritt ausgiebig getestet werden!

```
void *addr;
void *prev = nullptr;
const size_t page_size = 4096;
while ((addr = reinterpret_cast<void*>(alloc_page())) != nullptr) {
    // longmode.asm maps only the first 4 GiB
    if (reinterpret_cast<uintptr_t>(addr) >= 0x100000000)
        continue;

    DBG << "Checking " << addr << endl;

    // Fill full page with 0b01011010 pattern
    memset(addr, 0x5a, page_size);
    // Check contents of page with previous filled one
    assert(prev == nullptr || memcmp(prev, addr, page_size) == 0);

    prev = addr;
}
```

# Paging

---

Mit 64 bit sind (theoretisch) bis zu 16 EiB adressierbar

Mit 64 bit sind (theoretisch) bis zu 16 EiB adressierbar,  
aber der von uns verwendete x64 unterstützt „nur“

- maximal 52 bit (= 4 PiB) **physischen** Speicher

Mit 64 bit sind (theoretisch) bis zu 16 EiB adressierbar, aber der von uns verwendete x64 unterstützt „nur“

- maximal 52 bit (= 4 PiB) **physischen** Speicher
  - 40 bits durch Adressumsetzung + 12 bits Seite
  - MAXPHYADDR im Intel Manual (vgl. ISDMv3 4.1.4)

Mit 64 bit sind (theoretisch) bis zu 16 EiB adressierbar, aber der von uns verwendete x64 unterstützt „nur“

- maximal 52 bit (= 4 PiB) **physischen** Speicher
  - 40 bits durch Adressumsetzung + 12 bits Seite
  - MAXPHYADDR im Intel Manual (vgl. ISDMv3 4.1.4)

Mit 64 bit sind (theoretisch) bis zu 16 EiB adressierbar, aber der von uns verwendete x64 unterstützt „nur“

- maximal 52 bit (= 4 PiB) **physischen** Speicher
  - 40 bits durch Adressumsetzung + 12 bits Seite
  - MAXPHYADDR im Intel Manual (vgl. ISDMv3 4.1.4)
- standardmäßig 48 bit (= 256 TiB) **virtuellen** Speicher

Mit 64 bit sind (theoretisch) bis zu 16 EiB adressierbar, aber der von uns verwendete x64 unterstützt „nur“

- maximal 52 bit (= 4 PiB) **physischen** Speicher
  - 40 bits durch Adressumsetzung + 12 bits Seite
  - MAXPHYADDR im Intel Manual (vgl. ISDMv3 4.1.4)
- standardmäßig 48 bit (= 256 TiB) **virtuellen** Speicher
  - über 4-stufige Adressumsetzung

Mit 64 bit sind (theoretisch) bis zu 16 EiB adressierbar, aber der von uns verwendete x64 unterstützt „nur“

- maximal 52 bit (= 4 PiB) **physischen** Speicher
  - 40 bits durch Adressumsetzung + 12 bits Seite
  - MAXPHYADDR im Intel Manual (vgl. ISDMv3 4.1.4)
- standardmäßig 48 bit (= 256 TiB) **virtuellen** Speicher
  - über 4-stufige Adressumsetzung
  - die oberen 17 Bits einer Adresse müssen identisch sein (= *canonical*)  
→ valide Adressen sind 0x0 – 0x7fff ffff ffff sowie  
0xffff 8000 0000 0000 – 0xffff ffff ffff ffff

Mit 64 bit sind (theoretisch) bis zu 16 EiB adressierbar, aber der von uns verwendete x64 unterstützt „nur“

- maximal 52 bit (= 4 PiB) **physischen** Speicher
  - 40 bits durch Adressumsetzung + 12 bits Seite
  - MAXPHYADDR im Intel Manual (vgl. ISDMv3 4.1.4)
- standardmäßig 48 bit (= 256 TiB) **virtuellen** Speicher
  - über 4-stufige Adressumsetzung
  - die oberen 17 Bits einer Adresse müssen identisch sein (= *canonical*)  
→ valide Adressen sind 0x0 – 0x7fff ffff ffff sowie  
0xffff 8000 0000 0000 – 0xffff ffff ffff ffff
- neuere Architekturen 57 bit (= 128 PiB) **virt.** Speicher
  - über 5-stufige Adressumsetzung
  - muss extra aktiviert werden
  - die oberen 8 Bits müssen identisch sein (= *canonical*)

Für **STUBSMI** verwenden wir in dieser Aufgabe

- eine **Seitengröße von 4 KiB** wie im Beispiel  
(Hardware unterstützt 2 MiB sowie ggf. 1 GiB Seiten)

Für **STUBSMI** verwenden wir in dieser Aufgabe

- eine **Seitengröße von 4 KiB** wie im Beispiel  
(Hardware unterstützt 2 MiB sowie ggf. 1 GiB Seiten)
- eine **4-stufige Adressumsetzung** → 48 bit Adressen

Für **STUBSMI** verwenden wir in dieser Aufgabe

- eine **Seitengröße von 4 KiB** wie im Beispiel  
(Hardware unterstützt 2 MiB sowie ggf. 1 GiB Seiten)
- eine **4-stufige Adressumsetzung** → 48 bit Adressen
- bei 7.5 ECTS auch die Möglichkeit Seiten als **nicht-ausführbar** zu markieren  
(dafür muss im *Extended Feature Enable Register* [MSR\_EFER]  
das 11. Bit [MSR\_EFER\_NXE] gesetzt sein)

Für **STUBSMI** verwenden wir in dieser Aufgabe

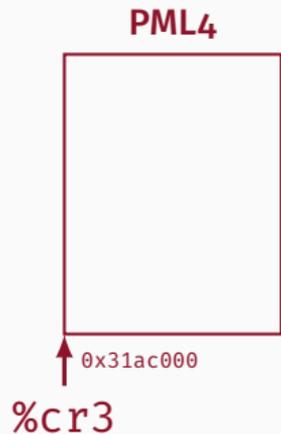
- eine **Seitengröße von 4 KiB** wie im Beispiel  
(Hardware unterstützt 2 MiB sowie ggf. 1 GiB Seiten)
- eine **4-stufige Adressumsetzung** → 48 bit Adressen
- bei 7.5 ECTS auch die Möglichkeit Seiten als **nicht-ausführbar** zu markieren  
(dafür muss im *Extended Feature Enable Register* [MSR\_EFER] das 11. Bit [MSR\_EFER\_NXE] gesetzt sein)
- **keine weiteren Features** wie *Protection Keys* (ISDMv3 4.6.2)

**Virtuelle Adresse:** `0x791bf4f2dafe`

Virtuelle Adresse: 0x791bf4f2dafe

%cr3

Virtuelle Adresse: 0x791bf4f2dafe



Virtuelle Adresse: 0x791bf4f2dafe

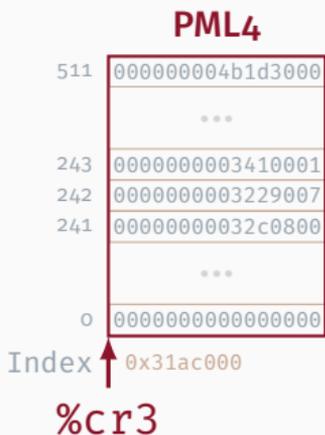
**PML<sub>4</sub>**

000000004b1d3000
...
000000003410001
000000003229007
0000000032c0800
...
0000000000000000

↑ 0x31ac000

**%cr3**

Virtuelle Adresse: 0x791bf4f2dafe



Virtuelle Adresse: 0x791bf4f2dafe

0111 1001 0001 1011 1111 0100 1111 0010 1101 1010 1111 1110

## PML<sub>4</sub>

511	000000004b1d3000
	...
243	0000000003410001
242	0000000003229007
241	00000000032c0800
	...
0	0000000000000000

Index ↑ 0x31ac000

%cr3

Virtuelle Adresse: 0x791bf4f2dafe

0111 1001 0001 1011 1111 0100 1111 0010 1101 1010 1111 1110

## PML<sub>4</sub>

511	000000004b1d3000
	...
243	0000000003410001
242	0000000003229007
241	00000000032c0800
	...
0	0000000000000000

Index ↑ 0x31ac000

%cr3

Virtuelle Adresse: 0x791bf4f2dafe

0111 1001 0001 1011 1111 0100 1111 0010 1101 1010 1111 1110

Bits 39 ... 47

**PML<sub>4</sub>**

511	000000004b1d3000
	...
243	0000000003410001
242	<b>0000000003229007</b>
241	00000000032c0800
	...
0	0000000000000000

Index ↑ 0x31ac000

**%cr3**

Virtuelle Adresse: 0x791bf4f2dafe

0111 1001 0001 1011 1111 1111 0100 1111 0010 1101 1010 1111 1110

## PML<sub>4</sub>

511	000000004b1d3000
	...
243	0000000003410001
242	0000000003229007
241	00000000032c0800
	...
0	0000000000000000

Index ↑ Base Flags

%cr3

Virtuelle Adresse: 0x791bf4f2dafe

0111 1001 0001 1011 1111 0100 1111 0010 1101 1010 1111 1110

**PML<sub>4</sub>**



0x3229000

0x31ac000

**%cr3**

242

Virtuelle Adresse: 0x791bf4f2dafa

0111 1001 0001 1011 1111 0100 1111 0010 1101 1010 1111 1110

PDP Table

PML<sub>4</sub>

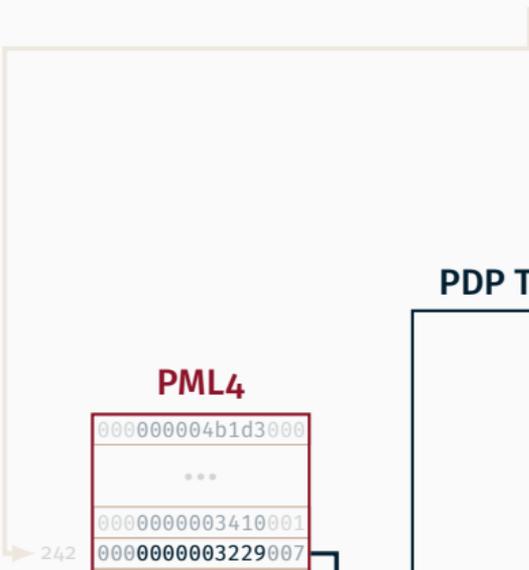
000000004b1d3000
...
000000003410001
000000003229007
0000000032c0800
...
0000000000000000

0x3229000

242

0x31ac000

%cr3



Virtuelle Adresse: 0x791bf4f2dafe

0111 1001 0001 1011 1111 0100 1111 0010 1101 1010 1111 1110

PDP Table

511	0000000000000000
	...
112	00000000035b7001
111	00000000035bb007
110	0000000000000000
	...
0	0000000003e11001

0x3229000

PML<sub>4</sub>

000000004b1d3000
...
0000000003410001
0000000003229007
00000000032c0800
...
0000000000000000

242

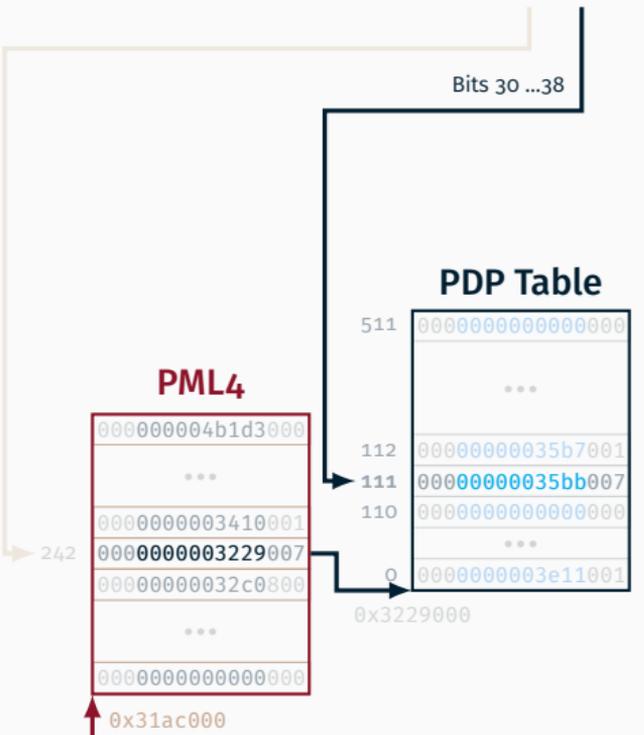
0x31ac000

%cr3

# 4-stufige Adressumsetzung (48 bit) am Beispiel

Virtuelle Adresse: 0x791bf4f2daffe

0111 1001 0001 1011 1111 0100 1111 0010 1101 1010 1111 1110



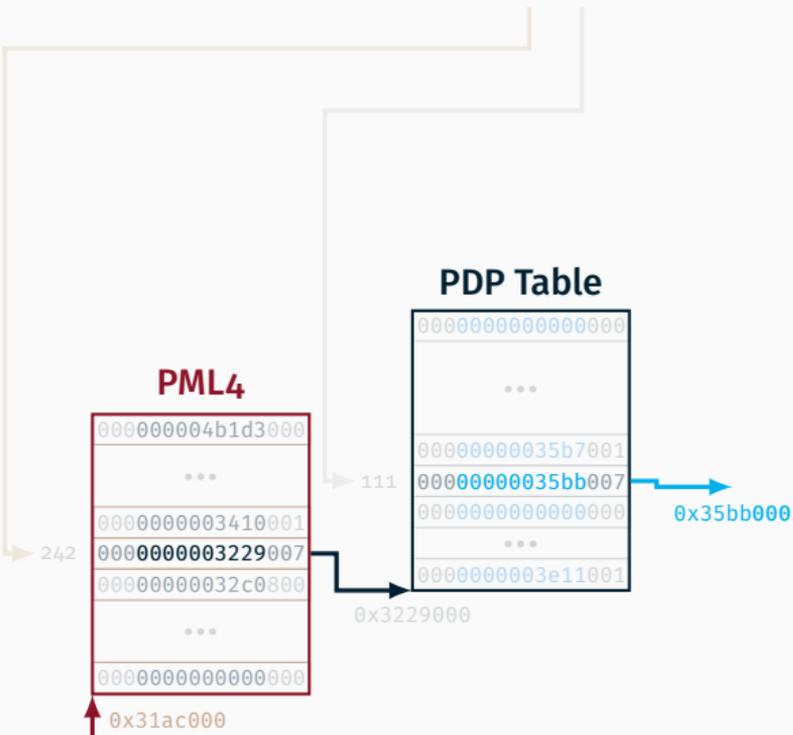
PDP Table

511	0000000000000000
...	...
112	00000000035b7001
111	00000000035bb007
110	0000000000000000
...	...
0	0000000003e11001

%cr3

Virtuelle Adresse: 0x791bf4f2dafe

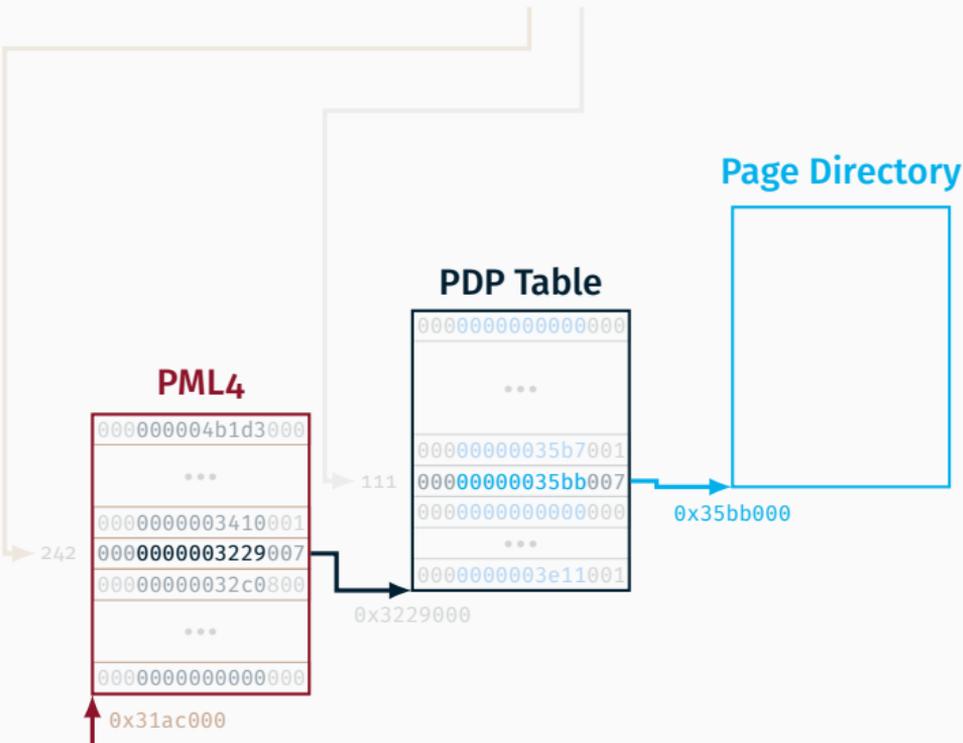
0111 1001 0001 1011 1111 0100 1111 0010 1101 1010 1111 1110



%cr3

Virtuelle Adresse: 0x791bf4f2daffe

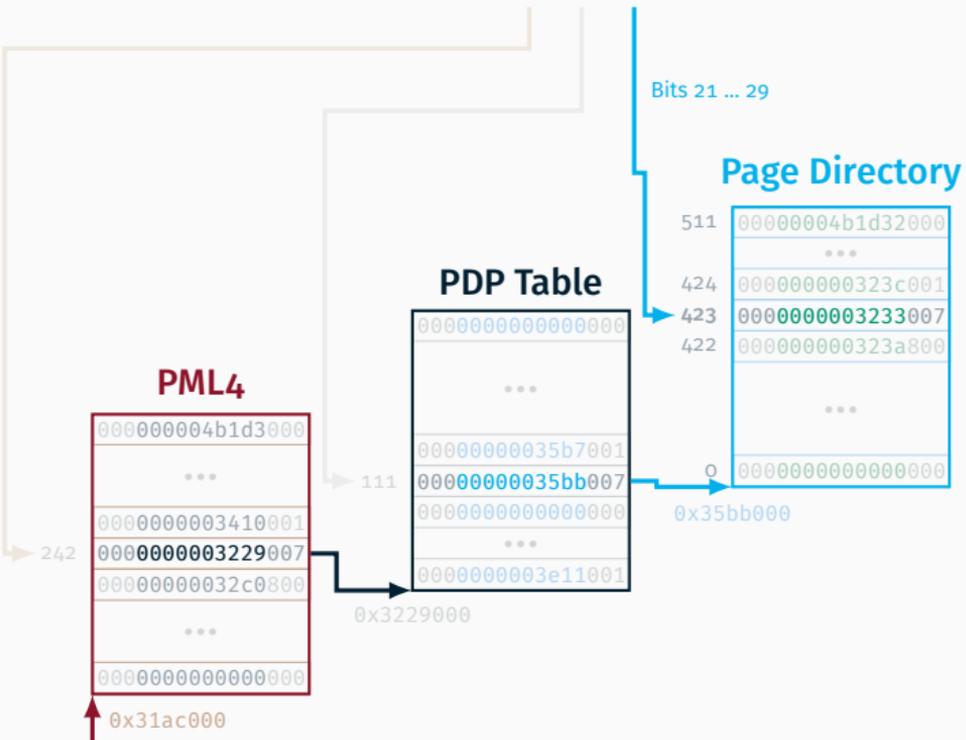
0111 1001 0001 1011 1111 0100 1111 0010 1101 1010 1111 1110



%cr3

Virtuelle Adresse: 0x791bf4f2dafa

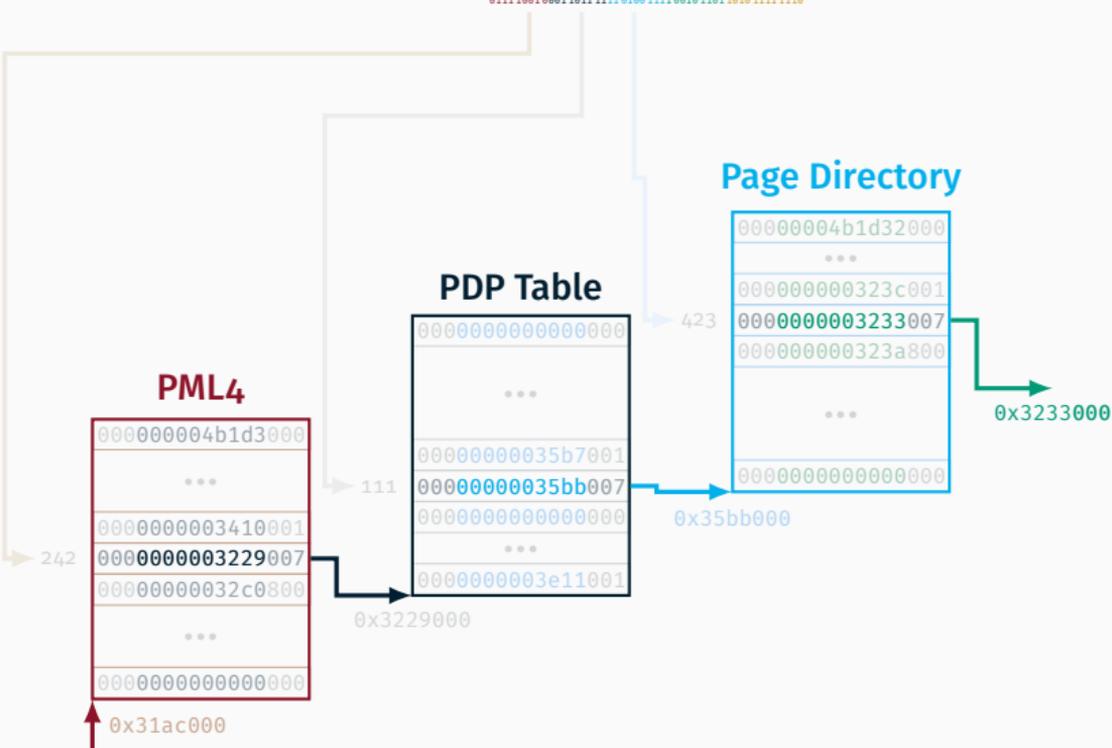
0111 1001 0001 1011 1111 0100 1111 0010 1101 1010 1111 1110



%cr3

Virtuelle Adresse: 0x791bf4f2daffe

0111 1001 0001 1011 1111 0100 1111 0010 1101 1010 1111 1110



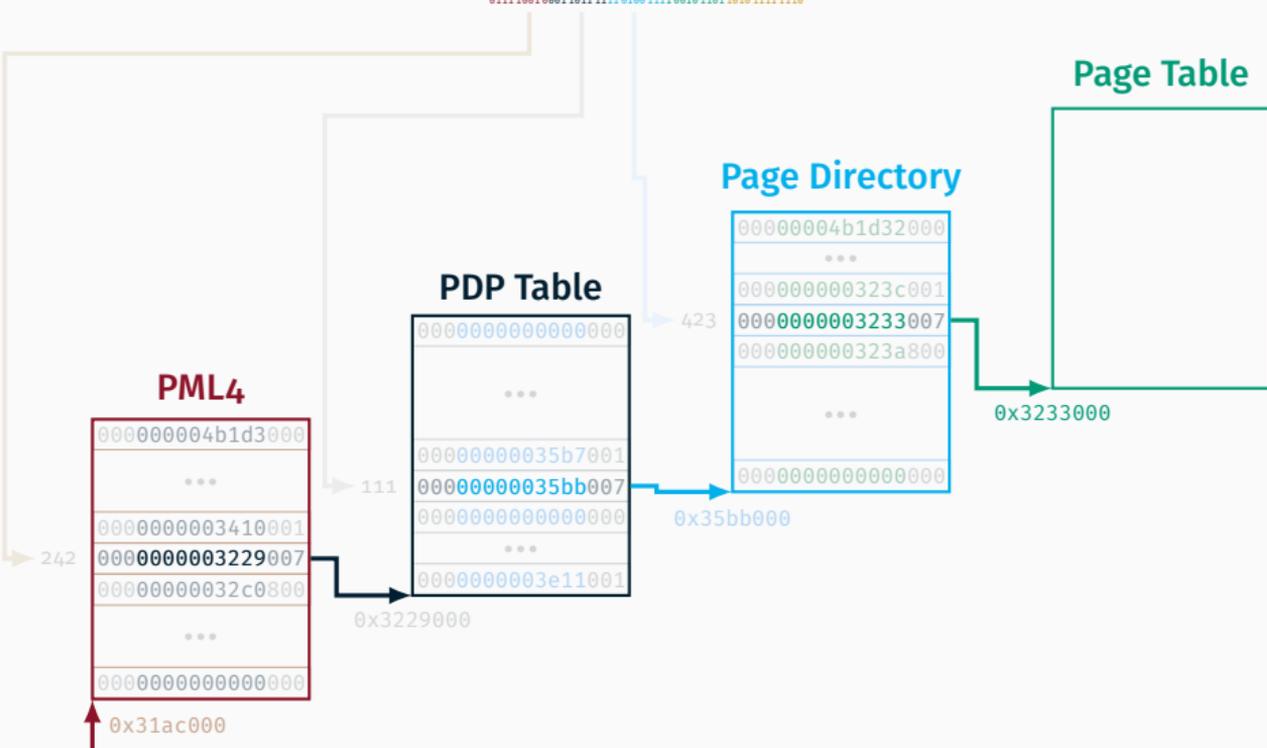
%cr3

# 4-stufige Adressumsetzung (48 bit) am Beispiel

ISDMv3 4.5

Virtuelle Adresse: 0x791bf4f2dafe

0111 1001 0001 1011 1111 0100 1111 0010 1101 1010 1111 1110



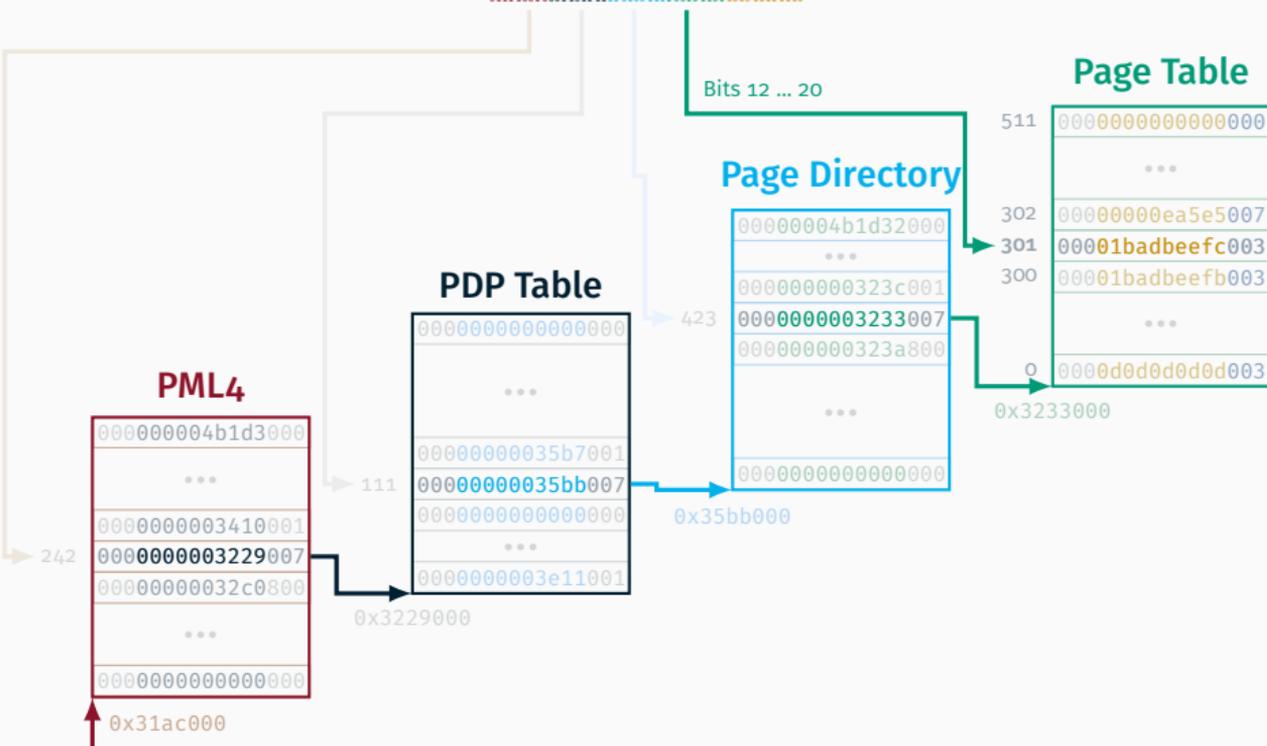
%cr3

# 4-stufige Adressumsetzung (48 bit) am Beispiel

ISDMv3 4.5

Virtuelle Adresse: 0x791bf4f2dafe

0111 1001 0001 1011 1111 0100 1111 0010 1101 1010 1111 1110



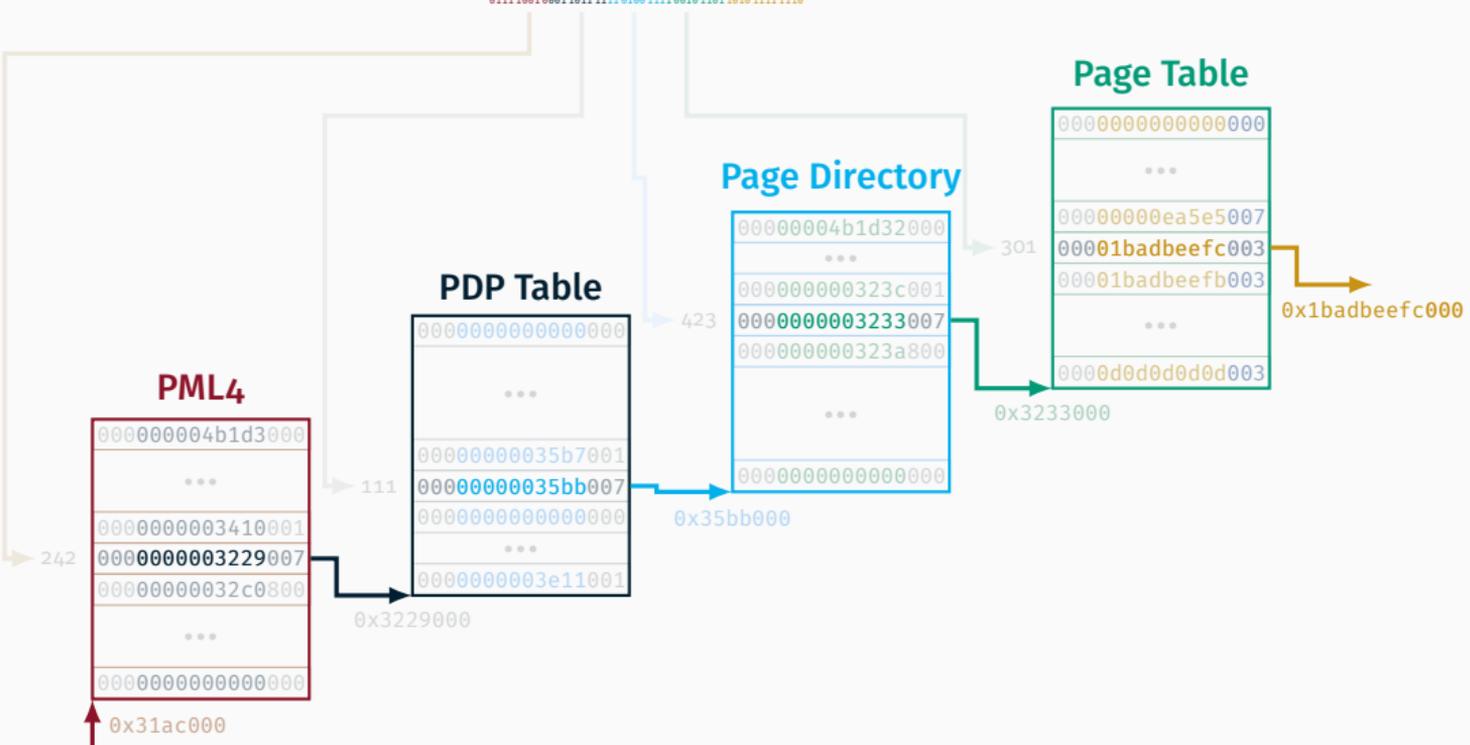
%cr3

# 4-stufige Adressumsetzung (48 bit) am Beispiel

ISDMv3 4.5

Virtuelle Adresse: 0x791bf4f2dafe

0111 1001 0001 1011 1111 0100 1111 0010 1101 1010 1111 1110



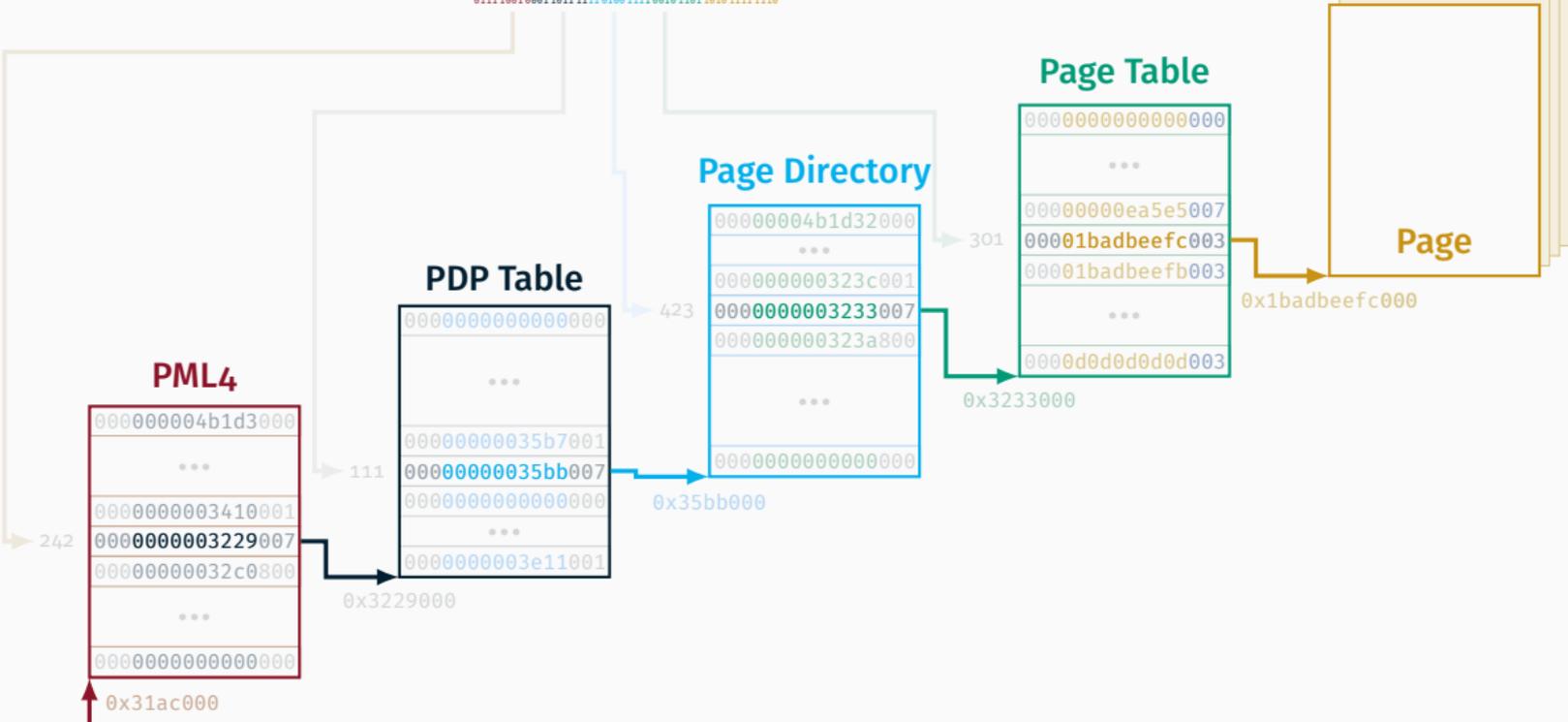
%cr3

# 4-stufige Adressumsetzung (48 bit) am Beispiel

ISDMv3 4.5

Virtuelle Adresse: 0x791bf4f2dafe

0111 1001 0001 1011 1111 0100 1111 0010 1101 1010 1111 1110



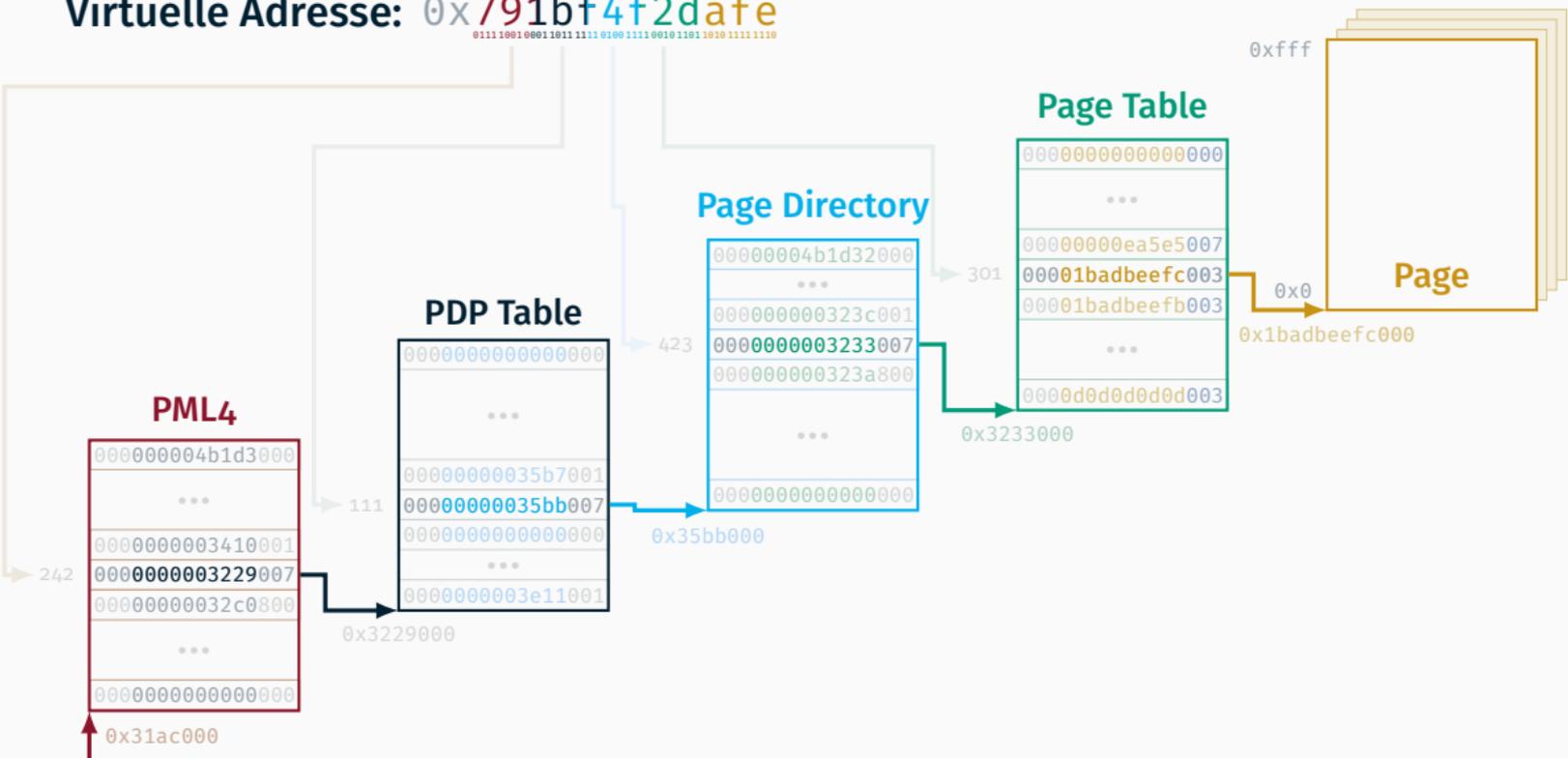
%cr3

# 4-stufige Adressumsetzung (48 bit) am Beispiel

ISDMv3 4.5

Virtuelle Adresse: 0x791bf4f2dafe

0111 1001 0001 1011 1111 0100 1111 0010 1101 1010 1111 1110



%cr3

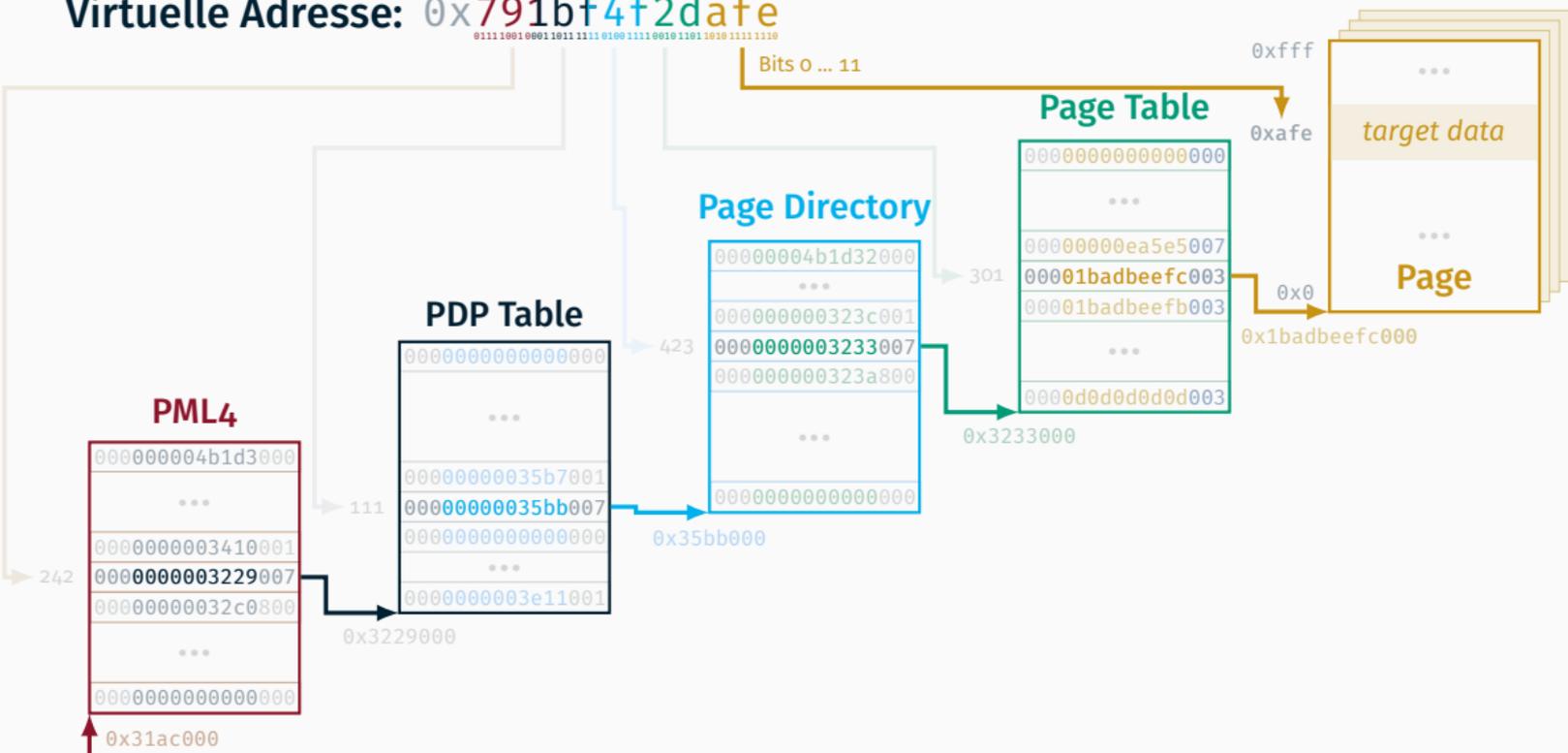
# 4-stufige Adressumsetzung (48 bit) am Beispiel

ISDMv3 4.5

Virtuelle Adresse: 0x791bf4f2dafe

0111 1001 0001 1011 1111 0100 1111 0010 1101 1010 1111 1110

Bits 0 ... 11



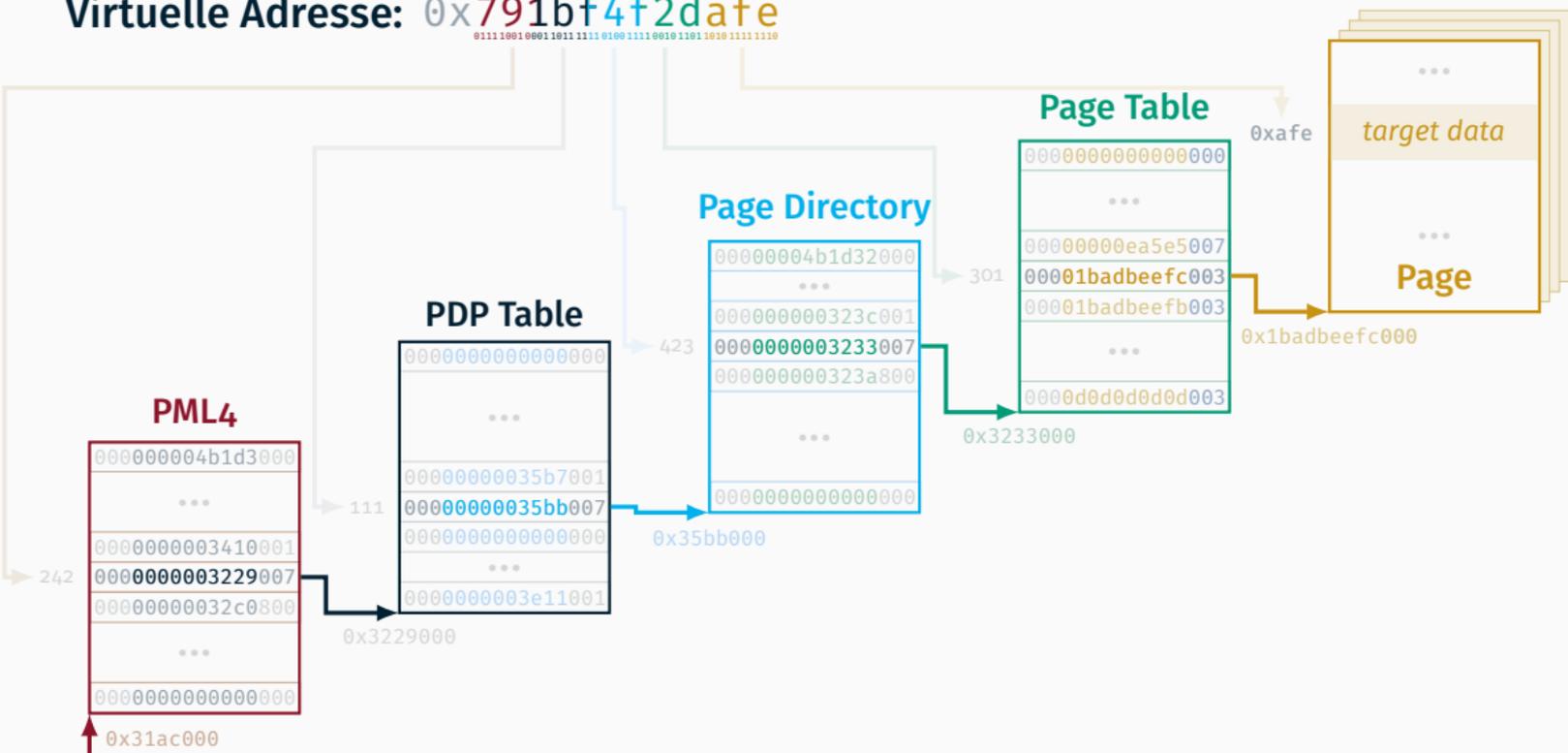
%cr3

# 4-stufige Adressumsetzung (48 bit) am Beispiel

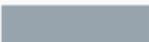
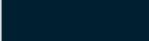
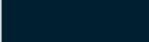
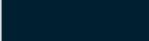
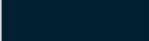
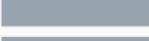
ISDMv3 4.5

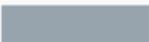
Virtuelle Adresse: 0x791bf4f2dafa

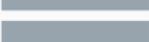
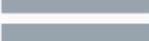
0111 1001 0001 1011 1111 0100 1111 0010 1101 1010 1111 1110

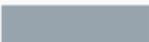


→ Physische Adresse: 0x1badbeefcafe

63		<b>Execute Disable:</b> 1 verhindert Ausführung von Code
62		<i>ignoriert</i>
52		
51		
		<b>(Physische) Adresse der <i>PDP Table</i>,</b> welche an einer 4 KiB-Grenze ausgerichtet sein muss
12		
11		<i>ignoriert</i>
9		
8		<b>Global:</b> Bei 4 KiB Seiten ignoriert
7		<i>reserviert, muss 0 sein</i>
6		<i>ignoriert</i>
5		<b>Accessed:</b> 1 falls die Zielseite verwendet wurde
4		<b>Page-Level Cache Disable:</b> 1 deaktiviert Caching
3		<b>Page-Level Write Through:</b> 1 aktiviert WT Caching
2		<b>User Mode:</b> 1 um Zugriff aus Ring 3 zu erlauben
1		<b>Writeable:</b> nur lesender (0) oder auch schreibender (1) Zugriff
0		<b>Present:</b> Eintrag aktiv (1) oder inaktiv (0)

63		<b>Execute Disable:</b> 1 verhindert Ausführung von Code
62		<i>ignoriert</i>
52		
51		<b>(Physische) Adresse des Seitenverzeichnisses (Page-Directory),</b> welche an einer 4 KiB-Grenze ausgerichtet sein muss
12		<i>ignoriert</i>
11		<i>ignoriert</i>
9		<b>Global:</b> Bei 4 KiB Seiten ignoriert
8		<b>Page Size:</b> Adresse zeigt auf Page-Directory (0) oder 1 GiB Seite (1)
7		
6		<i>ignoriert</i>
5		<b>Accessed:</b> 1 falls die Zielseite verwendet wurde
4		<b>Page-Level Cache Disable:</b> 1 deaktiviert Caching
3		<b>Page-Level Write Through:</b> 1 aktiviert WT Caching
2		<b>User Mode:</b> 1 um Zugriff aus Ring 3 zu erlauben
1		<b>Writeable:</b> nur lesender (0) oder auch schreibender (1) Zugriff
0		<b>Present:</b> Eintrag aktiv (1) oder inaktiv (0)

63		<b>Execute Disable:</b> 1 verhindert Ausführung von Code
62		<i>ignoriert</i>
52		
51		<b>(Physische) Adresse der Seitentabelle (Page Table),</b> welche an einer 4 KiB-Grenze ausgerichtet sein muss
12		
11		<i>ignoriert</i>
9		
8		<b>Global:</b> Bei 4 KiB Seiten ignoriert
7		<b>Page Size:</b> Adresse zeigt auf Page-Table (0) oder 2 MiB Seite (1)
6		<i>ignoriert</i>
5		<b>Accessed:</b> 1 falls die Zielseite verwendet wurde
4		<b>Page-Level Cache Disable:</b> 1 deaktiviert Caching
3		<b>Page-Level Write Through:</b> 1 aktiviert WT Caching
2		<b>User Mode:</b> 1 um Zugriff aus Ring 3 zu erlauben
1		<b>Writeable:</b> nur lesender (0) oder auch schreibender (1) Zugriff
0		<b>Present:</b> Eintrag aktiv (1) oder inaktiv (0)

63		<b>Execute Disable:</b> 1 verhindert Ausführung von Code
62		<i>ignoriert</i>
52		
51		<b>Physische Adresse</b> der 4 KiB Zielseite
12		<i>ignoriert</i>
11		<i>ignoriert</i>
9		
8		<b>Global:</b> Verhindert TLB Aktualisierung
7		<b>Page Attribute Table:</b> 1 aktiviert feingranulare Cacheeinstellung
6		<b>Dirty:</b> 1 falls auf die Zielseite geschrieben wurde
5		<b>Accessed:</b> 1 falls die Zielseite verwendet wurde
4		<b>Page-Level Cache Disable:</b> 1 deaktiviert Caching
3		<b>Page-Level Write Through:</b> 1 aktiviert WT Caching
2		<b>User Mode:</b> 1 um Zugriff aus Ring 3 zu erlauben
1		<b>Writeable:</b> nur lesender (0) oder auch schreibender (1) Zugriff
0		<b>Present:</b> Eintrag aktiv (1) oder inaktiv (0)

63	0	<b>Reserviert</b>
32		
31	1	<b>Paging</b> aktiv (1) oder inaktiv (0)
30	0	Cache Disable: 1 deaktiviert Caching
29	0	Not Write Through: 1 deaktiviert WT Caching
28		
19		<b>Reserviert</b>
18	0	Alignment Mask: 1 aktiviert Prüfung der Ausrichtung
17		<b>Reserviert</b>
16	0 / 1	<b>Write Protect:</b> 0 erlaubt schreiben in ro-Seiten im Ring 0
15		<b>Reserviert</b>
6		
5	0	Numeric Error: 1 aktiviert FPU Ausnahmebehandlung
4	0	Extension Type: für Koprozessor (Modelabhängig)
3	0	Task Switched
2	0	Emulation
1	0	Monitor Coprocessor
0	1	<b>Protection Enable:</b> Real (0) oder Protected (1) Mode

} für FPU Kontextsicherung



**Reserviert.** Für was auch immer.

63



0

## Page-Fault Linear Address

Beinhaltet bei einem Seitenfehler die virtuelle Adresse, die den Fehler verursacht hat.

63

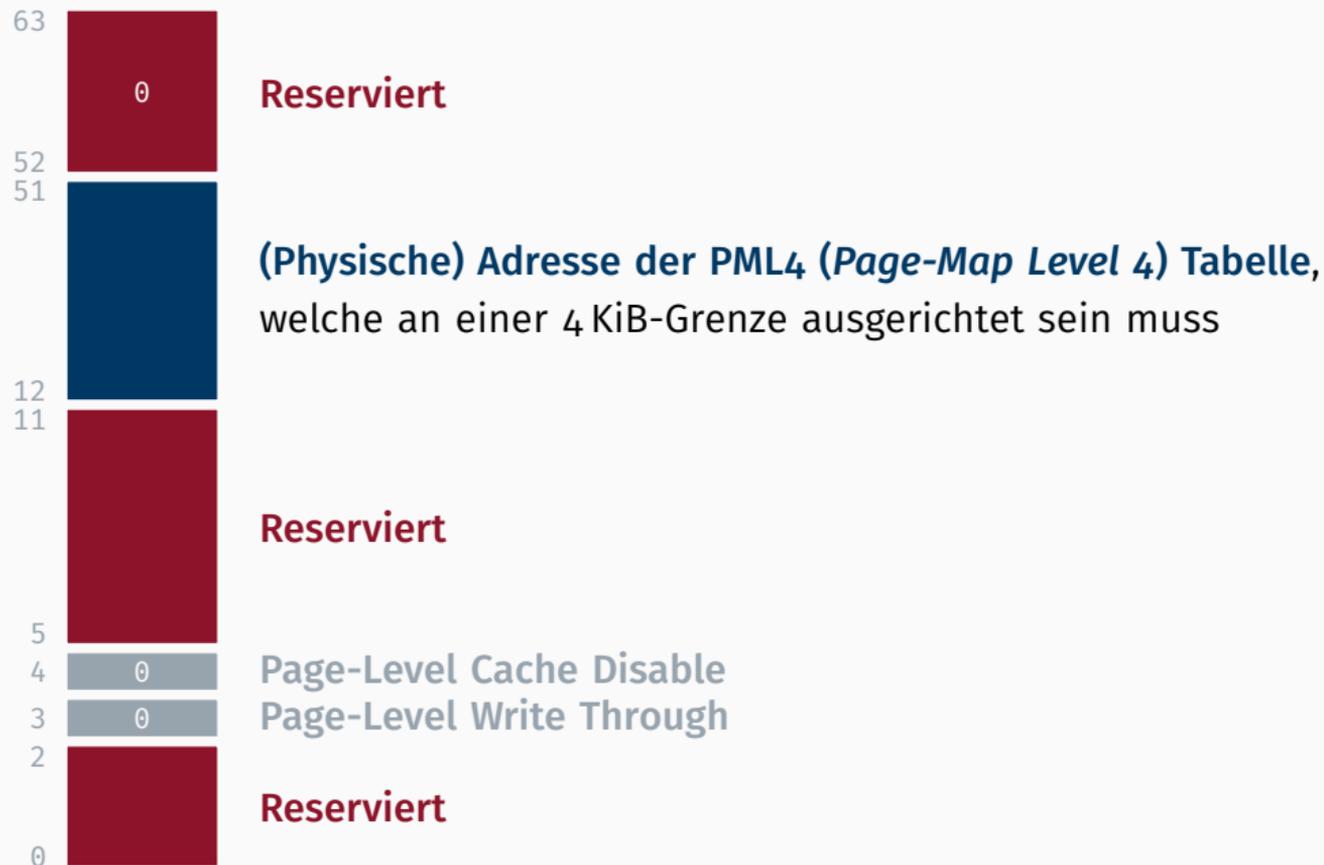


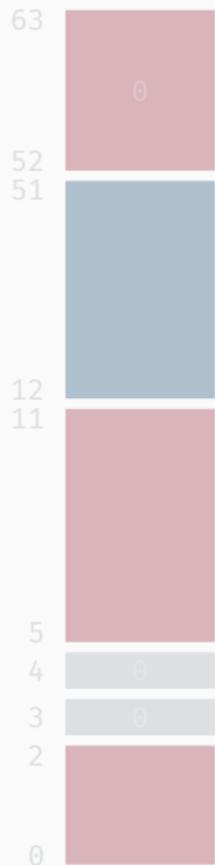
0

## Page-Fault Linear Address

Beinhaltet bei einem Seitenfehler die virtuelle Adresse, die den Fehler verursacht hat.

*(Noch) nicht notwendig in dieser Übung,  
aber kann das Entkäfern deutlich vereinfachen!*





Reserviert

(Physische) Adresse der PML4 (*Page-Map Level 4*) Tabelle, welche an einer 4 KiB-Grenze ausgerichtet sein muss



Reserviert

**Beim Schreiben von %cr3 wird TLB gespült**

Page-Level Cache Disable  
Page-Level Write Through

Reserviert

**%cr4** Steuerung von architekturabhängigen Erweiterungen wie **Page Size Extension** (4 MiB große Seiten) oder **Physical Address Extension** (erlaubt mehr als 4 GiB Speicher unter 32 Bit).

**%cr5** *reserviert*

**%cr6** *reserviert*

**%cr7** *reserviert*

**%cr8** steuert Zugriff auf *Task Priority Register*

**%cr4** Steuerung von architekturabhängigen Erweiterungen wie **Page Size Extension** (4 MiB große Seiten) oder **Physical Address Extension** (erlaubt mehr als 4 GiB Speicher unter 32 Bit).

**%cr5** *reserviert*

**%cr6** *reserviert*

**%cr7** *reserviert*

**%cr8** steuert Zugriff auf *Task Priority Register*

**Aber:** Nicht wichtig für uns, wir ignorieren diese in der Übung.

# Implementierungshinweise

- Einträge in den Tabellen als Struktur/Klasse abbilden
  - Methoden zum Nachschlagen nützlich (nachgebildete MMU)
  - 4 KiB Ausrichtung der Tabellen nicht vergessen

# Implementierungshinweise

- Einträge in den Tabellen als Struktur/Klasse abbilden
  - Methoden zum Nachschlagen nützlich (nachgebildete MMU)
  - 4 KiB Ausrichtung der Tabellen nicht vergessen
- Codeduplikation ist eine hervorragende Quelle für Leichtsinnsfehler
  - ggf. sind hier C++ Templates hilfreich
  - virtuelle Methoden nur mit Bedacht einsetzen  
(vtable vergrößert Struktur → `static_assert` ist hilfreich)

# Implementierungshinweise

- Einträge in den Tabellen als Struktur/Klasse abbilden
  - Methoden zum Nachschlagen nützlich (nachgebildete MMU)
  - 4 KiB Ausrichtung der Tabellen nicht vergessen
- Codeduplikation ist eine hervorragende Quelle für Leichtsinnsfehler
  - ggf. sind hier C++ Templates hilfreich
  - virtuelle Methoden nur mit Bedacht einsetzen  
(vtable vergrößert Struktur → `static_assert` ist hilfreich)
- die ersten 64 MB (Kernel space) sollen identitätsabgebildet sein
  - **Ausnahme:** erste Seite im Speicher (Adresse `0x0`) nicht mappen
  - in dieser Aufgabe auch noch aus Userspace les- & schreibbar
  - *für 7.5 ECTS:* Die Seiten mit `Kernel .text` müssen ausführbar sein.

# Implementierungshinweise

- Einträge in den Tabellen als Struktur/Klasse abbilden
  - Methoden zum Nachschlagen nützlich (nachgebildete MMU)
  - 4 KiB Ausrichtung der Tabellen nicht vergessen
- Codeduplikation ist eine hervorragende Quelle für Leichtsinnsfehler
  - ggf. sind hier C++ Templates hilfreich
  - virtuelle Methoden nur mit Bedacht einsetzen  
(vtable vergrößert Struktur → `static_assert` ist hilfreich)
- die ersten 64 MB (Kernelspace) sollen identitätsabgebildet sein
  - **Ausnahme:** erste Seite im Speicher (Adresse `0x0`) nicht mappen
  - in dieser Aufgabe auch noch aus Userspace les- & schreibbar
  - *für 7.5 ECTS:* Die Seiten mit `Kernel .text` müssen ausführbar sein.
- an im Speicher eingeblendete Geräte denken
  - entweder anderweitige Verwendung & Zugriff im Userspace verhindern
  - oder im Kernelspace einblenden

# Fragen?

---

Am 21.05. ist keine BST-Rechnerübung („bergfrei“)