Übung zu Betriebssystemtechnik

Aufgabe 3: Paging in StuBSmi

21. Mai 2025

Dustin Nguyen, Maximilian Ott & Phillip Raffeck

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

Lehrstuhl für Informatik 4
Systemsoftware

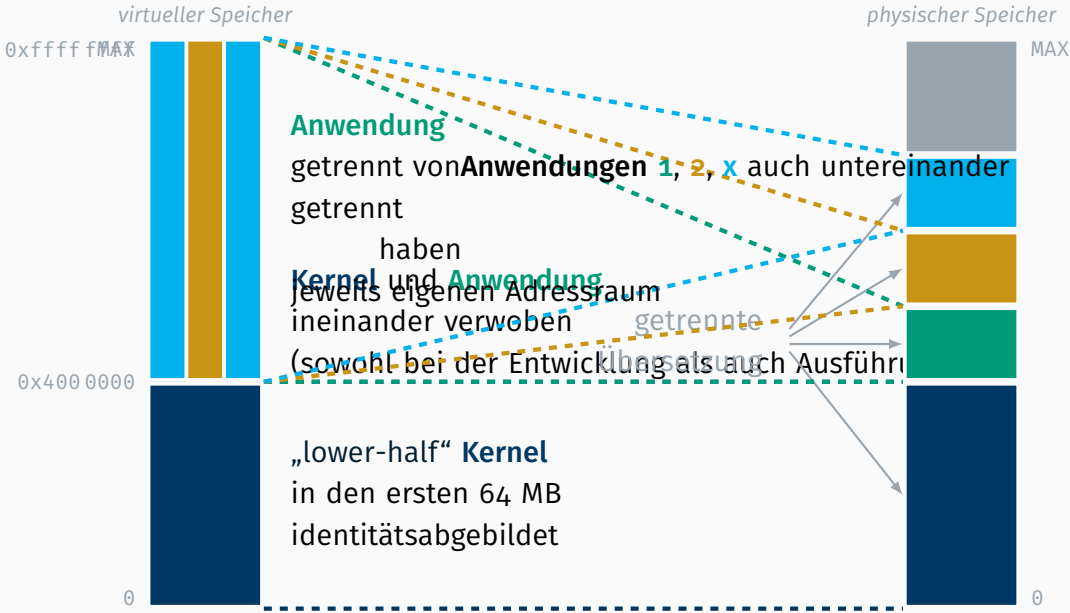
Friedrich-Alexander-Universität
Technische Fakultät

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)

Nach Aufgabe 4



Exkurs: Multiboot Specification *oder* Wie stiefel ich meinen Kernel?

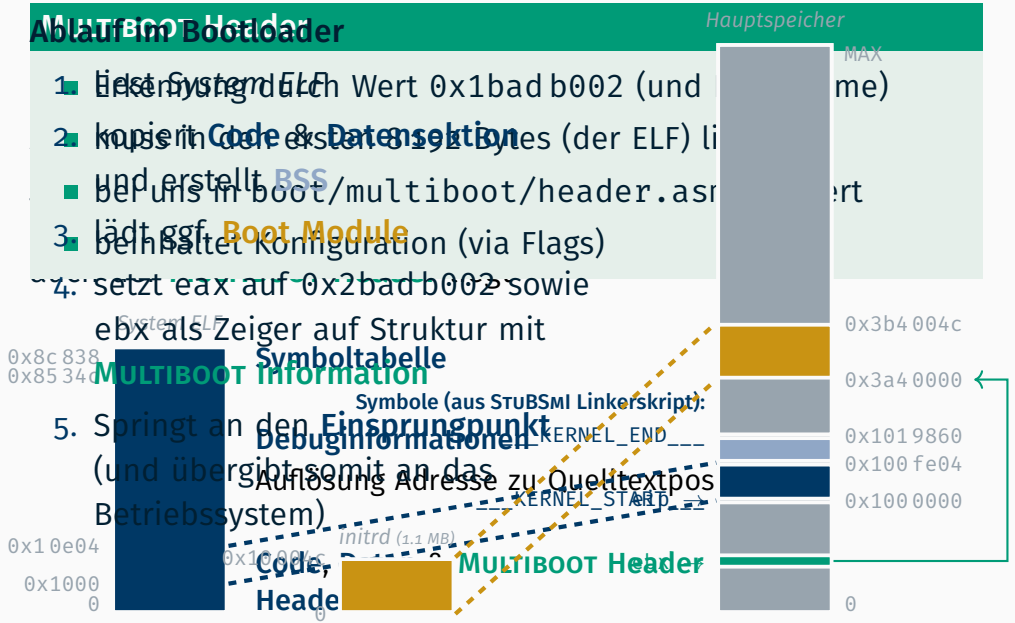
Überblick zur Multiboot Specification

- 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

Laden einer MULTIBOOT-kompatiblen Binärdatei

Ablauf im Bootloader

1. Erkennung durch Wert `0x1badb002` (und [illegible]me)
2. kopiert Code & Datensektion und erstellt BSS
   - bei uns in `boot/multiboot/header.asm` definiert
3. lädt ggf. Boot Module
4. setzt `eax` auf `0x2badb002` sowie `ebx` als Zeiger auf Struktur mit MULTIBOOT Information
5. Springt an den Einsprungpunkt (und übergibt somit an das Betriebssystem)



Multiboot Header in StubSmi

boot/multiboot/header.asm

```
[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)
```

Linkerskript in StuBSmi

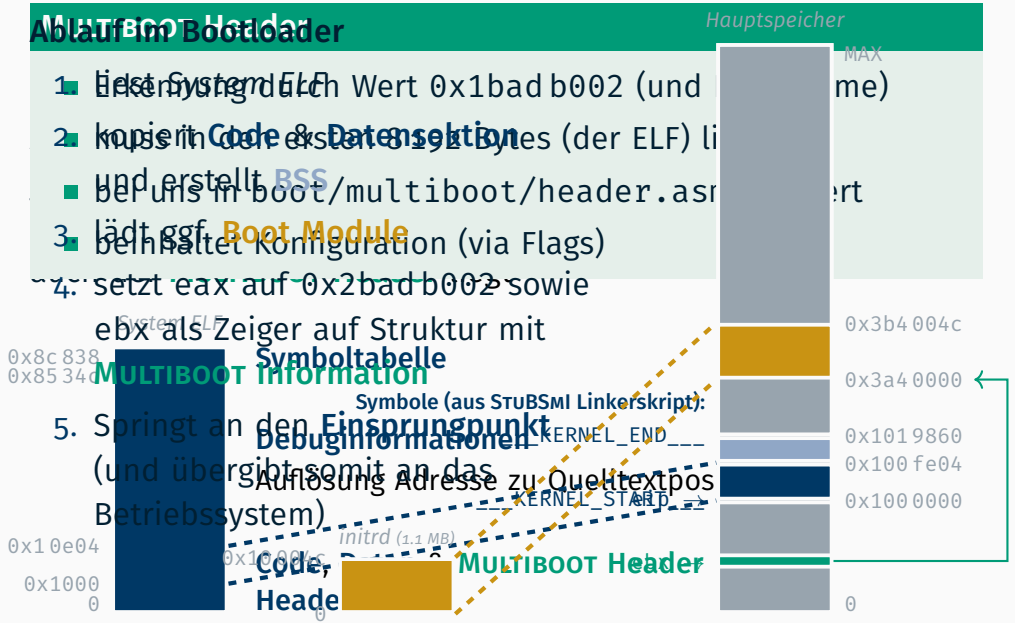
compiler/sections.ld

```
ENTRY(startup_bsp)                          /* entry function in our kernel */

SECTIONS {
  . = 16M;                                  /* start address of our kernel */
  ___KERNEL_START___ = .;                   /* custom source code reference */

  .boot : {
    *(.multiboot_header)                    /* should be at the very beginning */
  }

  .text : {
    *(.text .text.* .gnu.linkonce.t.*)      /* Kernel machine code */
    *(.init)
    *(.fini)
  }

  /* .setup_ap_seg, .rodata, (de-)initialization arrays, .data and .bss */

  ___KERNEL_END___ = .;                     /* another source code reference */
}
```

Laden einer MULTIBOOT-kompatiblen Binärdatei

Ablauf im Bootloader

1. Erkennung durch Wert `0x1badb002` (und [illegible]me)
2. kopiert Code & Datensektion und erstellt BSS
   - bei uns in `boot/multiboot/header.asm` definiert
3. lädt ggf. Boot Module
4. setzt `eax` auf `0x2badb002` sowie `ebx` als Zeiger auf Struktur mit MULTIBOOT Information
5. Springt an den Einsprungpunkt (und übergibt somit an das Betriebssystem)



Aufgabe 3

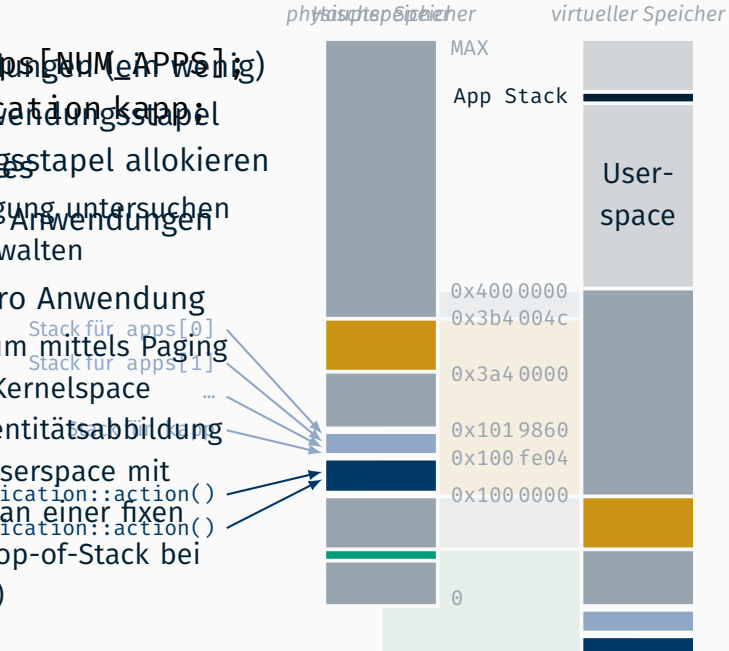
Aufgabeninhalt

Ziel dieser Übung: Anwendungen (ein wenig)

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

→

- dynamisch Anwendungsstapel allokieren
  - 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`)



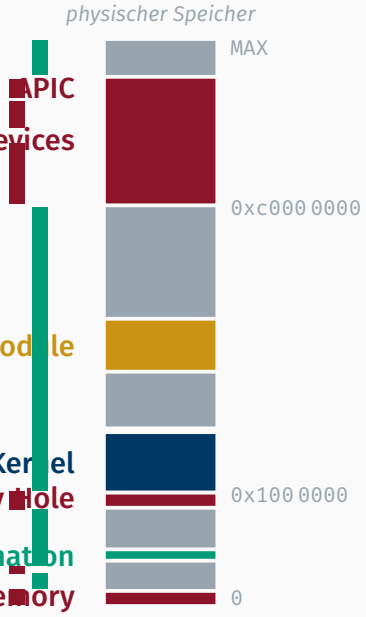
Speicherverwaltung

Freien Speicher finden

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!

```
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

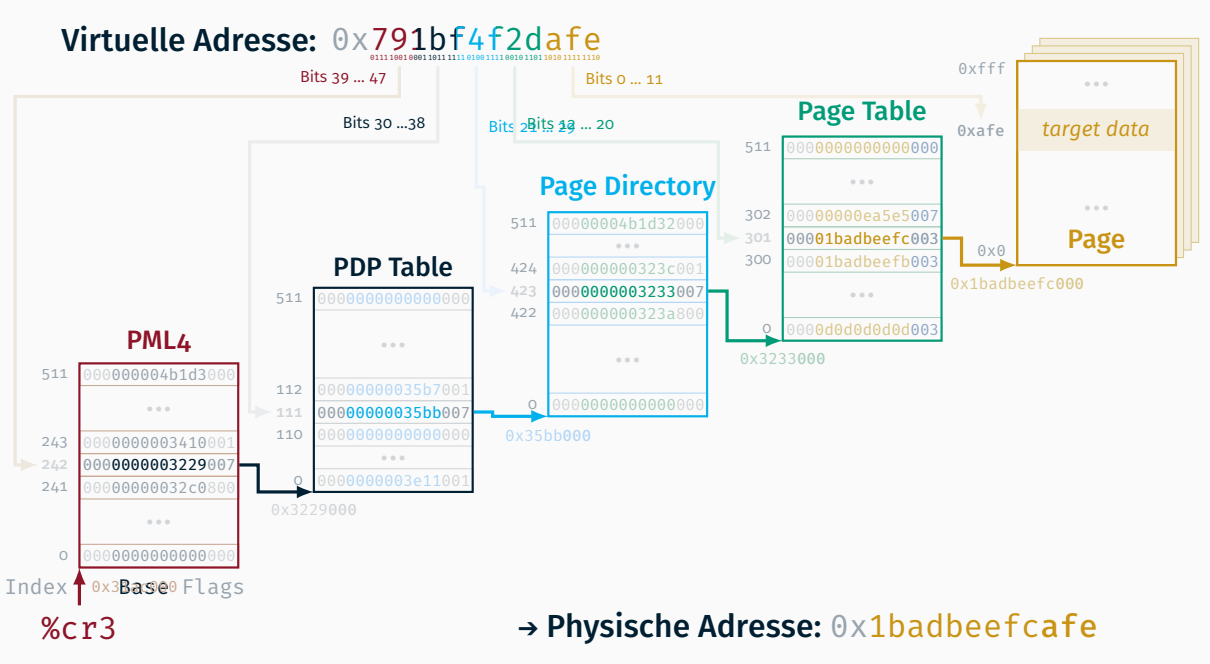
Mit 64 bit sind (theoretisch) bis zu 16 EiB adressierbar, aber der von uns verwendete x64 unterstützt „nur“

- 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)
- 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)

4-stufige Adressumsetzung (48 bit) am Beispiel



Eintrag im PML4 (Page-Map Level 4)

ISDMv3 Table 4-14

Execute Disable: 1 verhindert Ausführung von Code

ignoriert

(Physische) Adresse der *PDP Table*,
welche an einer 4 KiB-Grenze ausgerichtet sein muss

ignoriert

Global: Bei 4 KiB Seiten ignoriert

reserviert, muss 0 sein

ignoriert

Accessed: 1 falls die Zielseite verwendet wurde

Page-Level Cache Disable: 1 deaktiviert Caching

Page-Level Write Through: 1 aktiviert WT Caching

User Mode: 1 um Zugriff aus Ring 3 zu erlauben

Writeable: nur lesender (0) oder auch schreibender (1) Zugriff

Present: Eintrag aktiv (1) oder inaktiv (0)

Eintrag in der PDP (Page-Directory-Pointer) Table

ISDMv3 Table 4-16

Execute Disable: 1 verhindert Ausführung von Code

ignoriert

(Physische) Adresse des Seitenverzeichnisses (*Page-Directory*), welche an einer 4 KiB-Grenze ausgerichtet sein muss

ignoriert

Global: Bei 4 KiB Seiten ignoriert

Page Size: Adresse zeigt auf Page-Directory (0) oder 1 GiB Seite (1)

ignoriert

Accessed: 1 falls die Zielseite verwendet wurde

Page-Level Cache Disable: 1 deaktiviert Caching

Page-Level Write Through: 1 aktiviert WT Caching

User Mode: 1 um Zugriff aus Ring 3 zu erlauben

Writeable: nur lesender (0) oder auch schreibender (1) Zugriff

Present: Eintrag aktiv (1) oder inaktiv (0)

Eintrag im Seitenverzeichnis (Page-Directory)

ISDMv3 Table 4-18

Execute Disable: 1 verhindert Ausführung von Code

ignoriert

(Physische) Adresse der Seitentabelle (*Page Table*),
welche an einer 4 KiB-Grenze ausgerichtet sein muss

ignoriert

Global: Bei 4 KiB Seiten ignoriert

Page Size: Adresse zeigt auf Page-Table (0) oder 2 MiB Seite (1)

ignoriert

Accessed: 1 falls die Zielseite verwendet wurde

Page-Level Cache Disable: 1 deaktiviert Caching

Page-Level Write Through: 1 aktiviert WT Caching

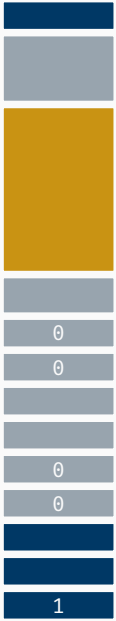
User Mode: 1 um Zugriff aus Ring 3 zu erlauben

Writeable: nur lesender (0) oder auch schreibender (1) Zugriff

Present: Eintrag aktiv (1) oder inaktiv (0)

Eintrag in der Seitentabelle (Page-Table)

ISDMv3 Table 4-19



Execute Disable: 1 verhindert Ausführung von Code

ignoriert

Physische Adresse der 4 KiB Zielseite

ignoriert

Global: Verhindert TLB Aktualisierung

Page Attribute Table: 1 aktiviert feingranulare Cacheeinstellung

Dirty: 1 falls auf die Zielseite geschrieben wurde

Accessed: 1 falls die Zielseite verwendet wurde

Page-Level Cache Disable: 1 deaktiviert Caching

Page-Level Write Through: 1 aktiviert WT Caching

User Mode: 1 um Zugriff aus Ring 3 zu erlauben

Writeable: nur lesender (0) oder auch schreibender (1) Zugriff

Present: Eintrag aktiv (1) oder inaktiv (0)

Kontrollregister 0 (`%cr0`)

Reserviert

Paging aktiv (`1`) oder inaktiv (`0`)

Cache Disable: 1 deaktiviert Caching

Not Write Through: 1 deaktiviert WT Caching

Reserviert

Alignment Mask: 1 aktiviert Prüfung der Ausrichtung

Reserviert

Write Protect: `0` erlaubt schreiben in `ro`-Seiten im Ring 0

Reserviert

Numeric Error: 1 aktiviert FPU Ausnahmebehandlung

Extension Type: für Koprozessor (Modelabhängig)

Task Switched

Emulation

Monitor Coprocessor

für FPU Kontextsicherung

Protection Enable: Real (`0`) oder Protected (`1`) Mode

Kontrollregister 1 (%cr1)

Reserviert. Für was auch immer.

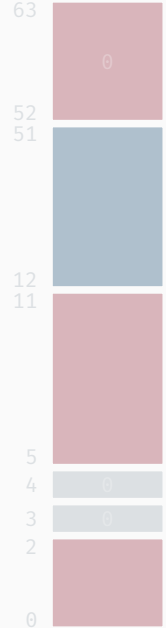
Kontrollregister 2 (`%cr2`)

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!

Kontrollregister 3 (%cr3)



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

weitere Kontrollregister

`%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
- Codeduplikation ist eine hervorragende Quelle für Leichtsinnsfehler
  - ggf. sind hier C++ Templates hilfreich
  - virtuelle Methoden nur mit Bedacht einsetzen
    (`vtable` vergrößert Strukur → `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“)