

# Betriebssystem-Entwicklung mit Literare Programming

Foliensatz 5: Booten, Protected Mode, Speicher



Wintersemester 2015/16

Dr. Hans-Georg Eßer

[h.g.esser@cs.fau.de](mailto:h.g.esser@cs.fau.de)  
<http://ohm.hgesser.de/>

v1.2, 04.10.2015

# (Organisatorisches)

- Auf Webseite ist Ulix-Buch verfügbar
- Ab der *übernächsten* Vorlesung: jeweils vorher Teile des Buchs zur Vorbereitung lesen
- Hinweise zur Lektüre immer in der vorherigen Vorlesung (und auf der Webseite)

# Booten (1)

- Allgemeiner Ablauf
  - PC führt BIOS-Code aus
  - BIOS-Routine sucht bootfähige Datenträger
  - BIOS lädt Bootsektor vom Datenträger und springt an Startadresse des Bootmanagers
  - Moderne Bootmanager („second stage boot loaders“) laden weiteren Code nach
  - Nach Auswahl lädt der Bootmanager Kernel und weitere Dateien (z. B. initrd) und springt an die Startadresse des Kernels

# Booten (2)

- Wie wir ULIX booten
  - Prinzipiell: möglich, eigenen Bootmanager zu schreiben
  - einfacher: GRUB verwenden
  - FAT-Diskette enthält GRUB, den Kernel (ulix.bin) und die Grub-Konfiguration (menu.lst)

```
<menu.lst>≡  
timeout 5
```

```
title ULIX-i386 (c) 2008-2013 F. Freiling & H.-G. Esser  
root (fd0)  
kernel /ulix.bin
```

# Booten (3)

- Multiboot-Spezifikation
  - GRUB erwartet, dass die Kernel-Datei am Anfang einen Multiboot-Header (12 Bytes) enthält:

|       |              |            |
|-------|--------------|------------|
| 00–03 | magic string | 0x1badb002 |
| 04–07 | flags        |            |
| 08–11 | checksum     |            |

- Flags: setze Bits 0 und 1 (load memory aligned, provide memory information to OS)
- Checksum:  $-(\text{magic} + \text{flags})$

# Booten (4)

```
<start.asm 68>≡
[section .setup]
[bits 32]
align 4
mboot:
    MB_HEADER_MAGIC    equ 0x1BADB002
    ; Header flags: page align (bit 0), memory info (bit 1)
    MB_HEADER_FLAGS    equ 11b    ; Bits: 1, 0
    MB_CHECKSUM         equ -(MB_HEADER_MAGIC + MB_HEADER_FLAGS)

    ; This is the GRUB Multiboot header. A boot signature
    dd MB_HEADER_MAGIC    ; 00..03: magic string
    dd MB_HEADER_FLAGS    ; 04..07: flags
    dd MB_CHECKSUM        ; 08..11: checksum
```

# Speicher

- Segmentierung (im Real Mode)
- Segmentierung (im Protected Mode)
- Vorbereitung auf Paging (virtuellen Speicher)
- Paging (→ später)

# Segmentierung: Real Mode (1)

- Beim PC-Start läuft der Rechner im Real Mode  
→ rückwärtskompatibel zum Intel 8086
- 16-Bit-Register  
→ maximal  $2^{16}$  Byte = 64 KByte adressierbar
- Durch Segmente 20-Bit-Adressen möglich  
→  $2^{20}$  Byte = 1 MByte adressierbar
- Segmentregister (CS, DS, ...) enthalten 16-Bit-Wert, der vier Bits nach links „geshiftet“ wird
- Zugriff auf  $x + DS \ll 4$  (statt  $x$ )



# Segmentierung: Real Mode (2)

- Beispiel: [1000:9abc]

DS = 0x1000

DS<<4 = 0x10000

adr = 0x09abc

Summe: 0x19abc

binär:

1 0000 0000 0000

1 0000 0000 0000 0000

0 1001 1010 1011 1100

1 0000 0000 0000 0000

1 1001 1010 1011 1100

- Mögliche Aufteilung des Speichers in 16 Segmente (16 x 64 KByte = 1 MByte):

[0000:0000]–[0000:FFFF], [1000:0000]–[1000:FFFF],  
[2000:0000]–[2000:FFFF], [3000:0000]–[3000:FFFF],

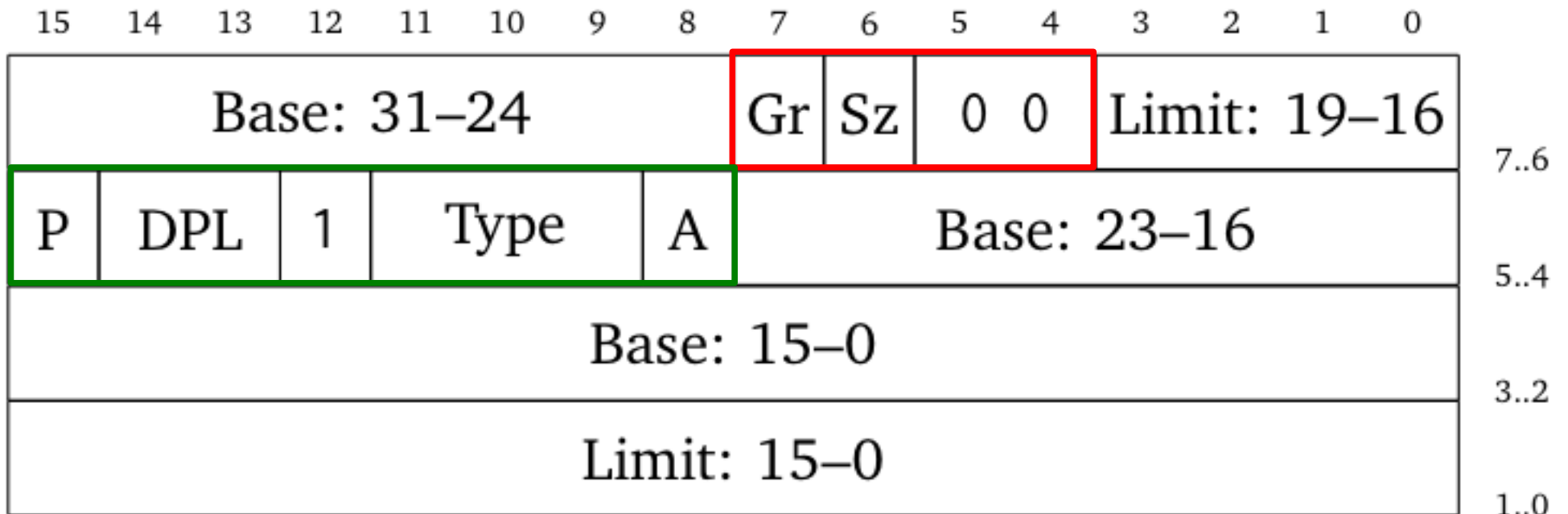
...

[C000:0000]–[C000:FFFF], [D000:0000]–[D000:FFFF],  
[E000:0000]–[E000:FFFF], [F000:0000]–[F000:FFFF]

# Segment.: Protected Mode (1)

- Im Protected Mode läuft Segmentierung über Segmentdeskriptoren
- CS, DS etc. enthalten nicht die Basisadresse des Segments, sondern Index in die Tabelle der Segmentdeskriptoren
  - Global Descriptor Table (GDT)
- Jeder Eintrag ist 8 Byte lang und enthält u. a. die Werte *Base* (32-bit) und *Limit* (20-bit)  
Index immer Vielfaches von 8 (0x08, 0x10, ...)

# Segment.: Protected Mode (2)



- Base und Limit nicht „am Stück“ gespeichert
- **Flags**: 1100 (Granularität: 4 KB, 32-bit-Deskr.)
- **Access Byte**: Zugriffsrechte

# Segm.: Protected Mode (3)

- **7: present bit**, must be set to 1
- **6/5: privilege level**, must be set to 00 for ring 0 (kernel mode) or 11 (=3) for ring 3 (user mode)
- **4:** reserved, must contain 1
- **3: executable bit**, we will set this to 1 in our code segment descriptor and to 0 in our data segment descriptor
- **2: direction bit / conforming bit:** for the data segment, 0 means that the segment grows upwards; for the code segment, 0 means that the code in this segment can only be executed if the CPU operates in the ring that is declared in bits 6/5 (privilege level)
- **1: readable bit / writable bit:** we always set these to 1; for a code segment it means that we can also read from this segment, and for a data segment it means we can also write to it.
- **0: accessed bit:** we set this to 0; the CPU flips it to 1 when this segment is accessed.

# Segm.: Protected Mode (4)

- Wir brauchen zwei Einträge (Code, Daten)
  - `10011010` for the **code** segment  
`pR01x!ra`  
(present; ring 0; fixed-1; executable; exact privilege level; allow reading; not accessed)
  - `10010010` for the **data** segment  
`pR01x^wa`  
(present; ring 0; fixed-1; not executable; grow upwards; allow writing; not accessed).

# Vorbereitung für Paging (1)

- Gewünschte Speicheraufteilung
- Kernel so kompilieren, dass er Adressen ab `0xC0000000` verwendet
- aber wohin laden?  
→ am Anfang ist Paging nicht aktiviert

|                               |              |
|-------------------------------|--------------|
| 0xFFFFFFFF<br>⋮<br>0xC0000000 | Kernel space |
| 0xBFFFFFFF<br>⋮<br>0x00000000 | User space   |

# Vorbereitung für Paging (2)

- Trick: Segment-Deskriptoren mit Base-Adresse `0x40000000` erzeugen
- Kernel-Code ab Adresse `0xC0010000` erzeugen
- Beispiel:

- $$\begin{array}{r} 0xC0010ABC \\ + 0x40000000 \\ = 0x100010ABC \end{array}$$

↑  
Übertrag (>32 bit), fällt weg

# Vorbereitung für Paging (3)

- Also Base: 0x40000000; Limit im Prinzip egal – wir setzen es auf 0xFFFFFFFF

`<start.asm>+≡`

```
trickgdt:
    dw gdt_end - gdt_data - 1          ; GDT size
    dd gdt_data                        ; linear address of GDT

gdt_data:
    ; selector 0x00: empty entry
    dd 0, 0
    ; code selector 0x08 (code segment):
    db 0xFF, 0xFF, 0x00, 0x00, 0x00, 10011010b, 11001111b, 0x40
    ; data selector 0x10 (data segment):
    db 0xFF, 0xFF, 0x00, 0x00, 0x00, 10010010b, 11001111b, 0x40
gdt_end:
```

Limit  
Base



# Vorbereitung für Paging (4)

## Laden der Deskriptor-Tabelle:

`<start.asm> +≡`

```
[section .setup]
```

```
start:
```

```
    lgdt [trickgdt]
```

```
    mov ax, 0x10
```

```
    mov ds, ax
```

```
    mov es, ax
```

```
    mov fs, ax
```

```
    mov gs, ax
```

```
    mov ss, ax
```

```
    ; far jump
```

```
    jmp 0x08:higherhalf
```

```
[section .text]
```

```
higherhalf:
```

```
    ; ab hier im prot. mode
```

Section .setup: vom  
Assembler so erzeugt, dass  
sie niedrige Adressen (ab  
0x10000) verwendet

(0x08: code, 0x10: data)


Section .text: weiterer  
Assembler-Code und C-  
Kernel – mit Adressen ab  
0xC0100000

# Aufbau der Kernel-Quellen

- schon gesehen: `start.asm`
  - viel mehr passiert dort nicht; es fehlt nur Code für Interrupt- und Exception-Behandlung
- der meiste Code steht in `ulix.c`
  - ca. 97% C-Code
  - teilweise Inline-Assembler
- daneben noch zwei C-Dateien
  - `printf.c` (externe Implementierung von `printf`)
  - `module.c` (→ später)


# Aufbau der C-Datei `ulix.c` (1)

```
<ulix.c 77> ≡  
  /* <copyright notice 56> */  
  <constants 93b>  
  <macro definitions 63a>  
  <elementary type definitions 148b>  
  <type definitions 70a>  
  <function prototypes 78d>  
  <global variables 71>  
  <function implementations 79c>  
  <kernel main 78a>
```



# Aufbau der C-Datei `u1ix.c` (2)

```
<kernel main 78a> ≡  
int main (void *mboot_ptr, unsigned int initial_stack) {  
    <initialize kernel global variables 219b>  
    <setup serial port 418a> // for debugging  
    <setup memory 78b> ←  
    <setup video 78c>  
    <initialize system 79a>  
    <initialize syscalls 116c>  
    <initialize filesystem 79b>  
    initialize_module(); // external code  
    <start shell 79d>  
}
```

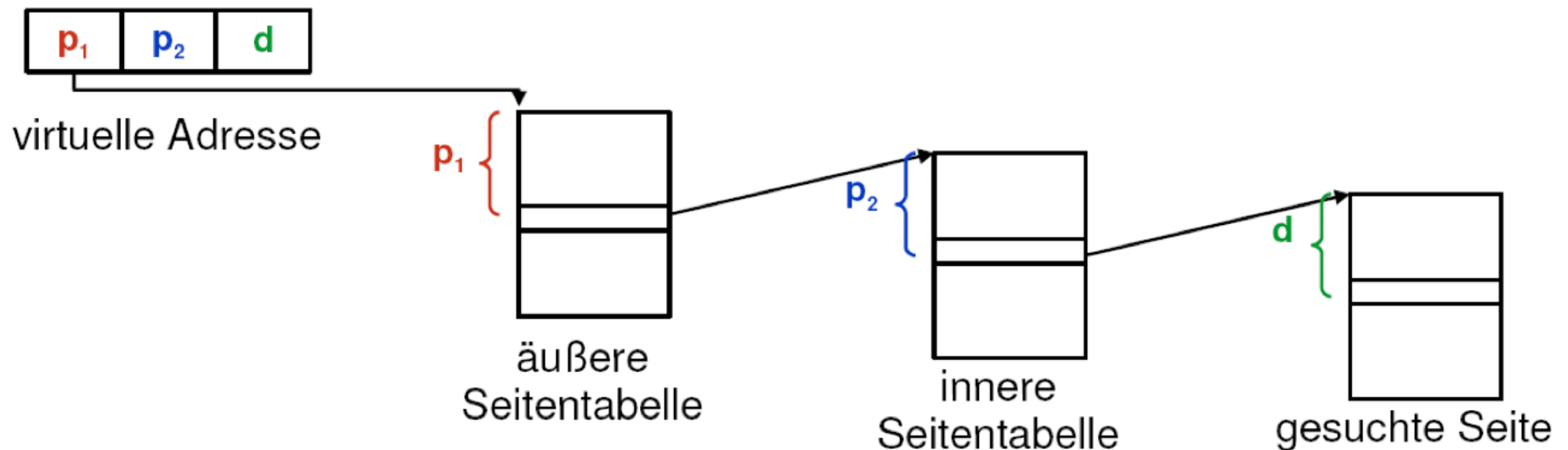


# Paging aktivieren

```
<setup memory 78b>≡  
  <setup identity mapping for kernel 161a>  
  <enable paging for the kernel 161b>  
  gdt_install();
```

# Intel: Seitentabellen (1)

- allgemein mehrstufige Seitentabellen
- z. B. zweistufig:

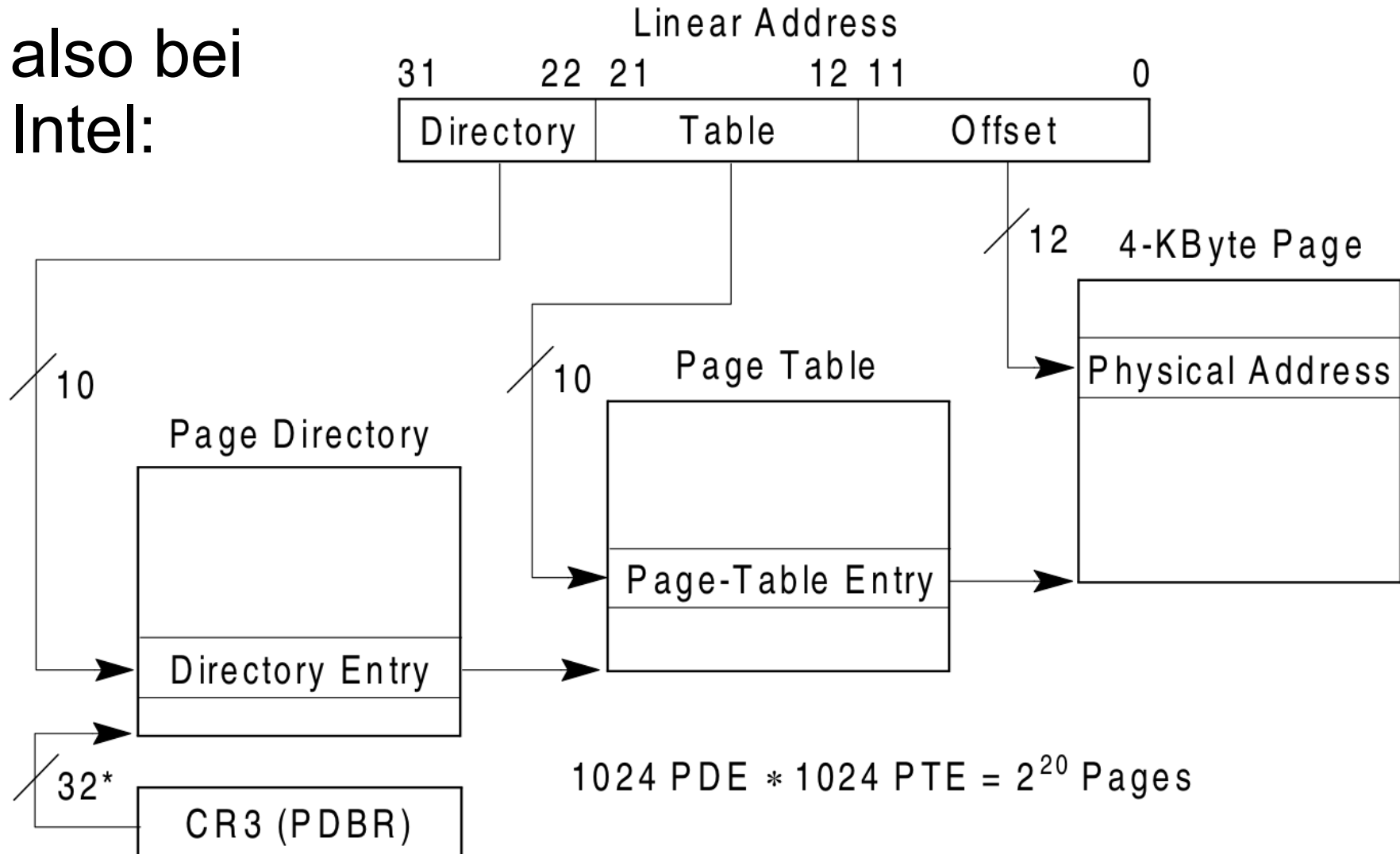


- Intel-Notation:

**Page Directory** (außen), **Page Table** (innen)

# Intel: Seitentabellen (2)

- also bei Intel:



Quelle: Intel Architecture Software Developer's Manual  
Volume 3: System Programming Guide, p. 3-20

# Intel: Seitentabellen (3)

- Ein Eintrag des Page Directory (außen) heißt
  - **Page Directory Entry** oder
  - **Page Table Descriptor**  
(zeigt auf eine Page Table)
- Ein Eintrag der Page Table (innen) heißt
  - **Page Table Entry** oder
  - **Page Descriptor** (zeigt auf einen Seitenrahmen)
- Aufbau der beiden Datentypen fast gleich

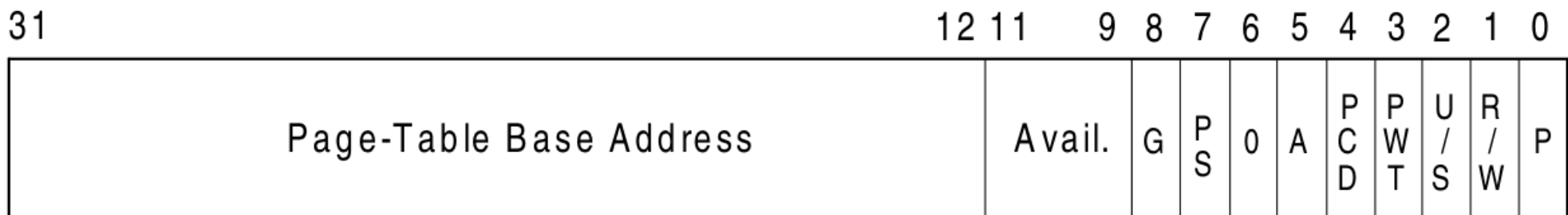


# Intel: Seitentabellen (4)

- Phys. Speicher ist in **Page Frames** der Größe 4 KB unterteilt
- Auch Page Directories und Page Tables sind 4 KB groß, passen also genau in einen Frame
- Anfang eines Frames (auch: eines Page Directorys, einer Page Table) immer ein Vielfaches von 4 KB
- darum reichen zum Speichern der phys. Adresse die obersten 20 Bit eines 32-Bit-Adresse aus (12 Bit  $\rightarrow 2^{12}$  Byte = 4 KB)

# ULIX: Seitentabellen (1)

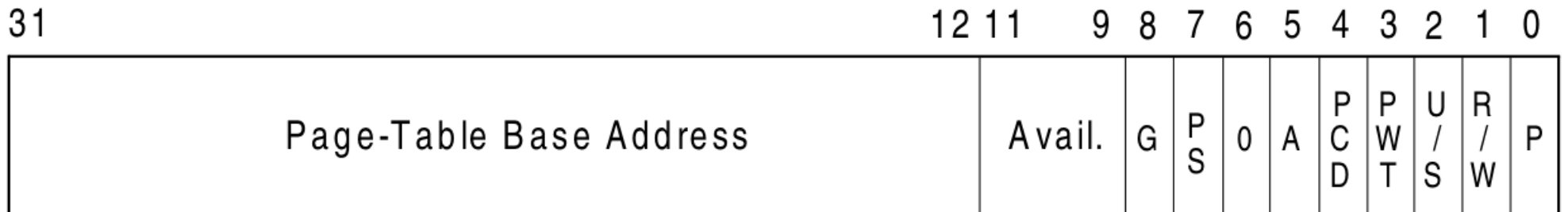
- Page Table Descriptor `page_table_desc`
  - ist 32 Bit groß
  - enthält in den oberen 20 Bit die oberen 20 Bit der phys. Adresse einer Page Table
  - enthält in den unteren 12 Bit Attribute
  - vollständige phys. Adresse berechnen = untere 12 Bit auf 0 setzen



# ULIX: Seitentabellen (2)

*{ type declarations } + ≡*

```
typedef struct {
    unsigned int present      : 1; // 0
    unsigned int writeable   : 1; // 1
    unsigned int user_accessible : 1; // 2
    unsigned int pwt         : 1; // 3 (page write transparent)
    unsigned int pcd         : 1; // 4 (page cache disabled)
    unsigned int accessed    : 1; // 5
    unsigned int undocumented : 1; // 6
    unsigned int zeroes      : 2; // 8.. 7
    unsigned int unused_bits  : 3; // 11.. 9
    unsigned int frame_addr   : 20; // 31..12
} page_table_desc;
```



# ULIX: Seitentabellen (3)

- Page Descriptor `page_desc`
  - ist 32 Bit groß
  - enthält in den oberen 20 Bit die oberen 20 Bit der phys. Adresse eines Page Frames
  - enthält in den unteren 12 Bit Attribute
  - vollständige phys. Adresse berechnen  
= untere 12 Bit auf 0 setzen



# ULIX: Seitentabellen (4)

```

<type declarations>+≡
typedef struct {
    unsigned int present           : 1; // 0
    unsigned int writeable        : 1; // 1
    unsigned int user_accessible  : 1; // 2
    unsigned int pwt              : 1; // 3
    unsigned int pcd              : 1; // 4
    unsigned int accessed         : 1; // 5
    unsigned int dirty            : 1; // 6
    unsigned int zeroes          : 2; // 8.. 7
    unsigned int unused_bits     : 3; // 11.. 9
    unsigned int frame_addr      : 20; // 31..12
} page_desc;

```

31

12 11 9 8 7 6 5 4 3 2 1 0



# ULIX: Seitentabellen (5)

## Datenstrukturen im Vergleich

```
typedef struct {
    unsigned int present           : 1; // 0
    unsigned int writeable        : 1; // 1
    unsigned int user_accessible  : 1; // 2
    unsigned int pwt              : 1; // 3
    unsigned int pcd              : 1; // 4
    unsigned int accessed         : 1; // 5
    unsigned int undocumented     : 1; // 6
    unsigned int zeroes           : 2; // 8.. 7
    unsigned int unused_bits      : 3; // 11.. 9
    unsigned int frame_addr      : 20; // 31..12
} page_table_desc;
```

```
typedef struct {
    unsigned int present           : 1;
    unsigned int writeable        : 1;
    unsigned int user_accessible  : 1;
    unsigned int pwt              : 1;
    unsigned int pcd              : 1;
    unsigned int accessed         : 1;
    unsigned int dirty            : 1;
    unsigned int zeroes           : 2;
    unsigned int unused_bits      : 3;
    unsigned int frame_addr      : 20;
} page_desc;
```

# UNIX: Seitentabellen (6)

- Page Table Descriptor mit Inhalt füllen:

*{function implementations}+≡*

```
page_table_desc* fill_page_table_desc (page_table_desc *ptd,  
    unsigned int present, unsigned int writeable,  
    unsigned int user_accessible, unsigned int frame_addr) {  
  
    // first fill the four bytes with zeros  
    memset (ptd, 0, sizeof(ptd));  
  
    // now enter the argument values in the right elements  
    ptd->present = present;  
    ptd->writeable = writeable;  
    ptd->user_accessible = user_accessible;  
    ptd->frame_addr = frame_addr >> 12;    // right shift, 12 bits  
    return ptd;  
};
```

# UNIX: Seitentabellen (7)

- Page Descriptor mit Inhalt füllen:

*<function implementations>+≡*

```
page_desc* fill_page_desc (page_desc *pd, unsigned int present,
    unsigned int writeable, unsigned int user_accessible,
    unsigned int dirty, unsigned int frame_addr) {

    // first fill the four bytes with zeros
    memset (pd, 0, sizeof(pd));

    // now enter the argument values in the right elements
    pd->present = present;
    pd->writeable = writeable;
    pd->user_accessible = user_accessible;
    pd->dirty = dirty;
    pd->frame_addr = frame_addr >> 12;    // right shift, 12 bits
    return pd;
};
```



# ULIX: Seitentabellen (8)

- Makros für einfacheren Aufruf
  - für Page Table Descriptors:

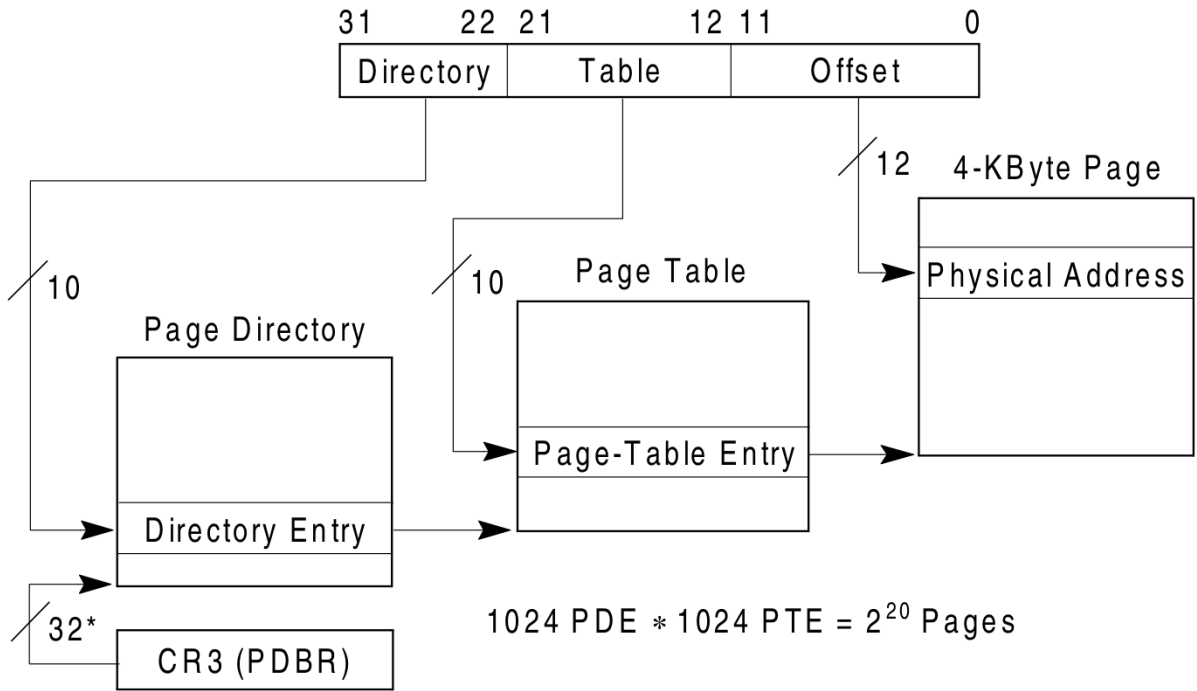
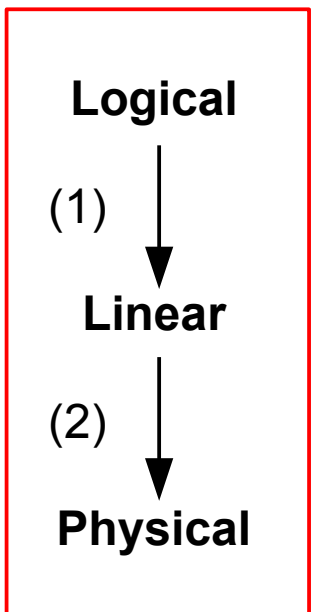
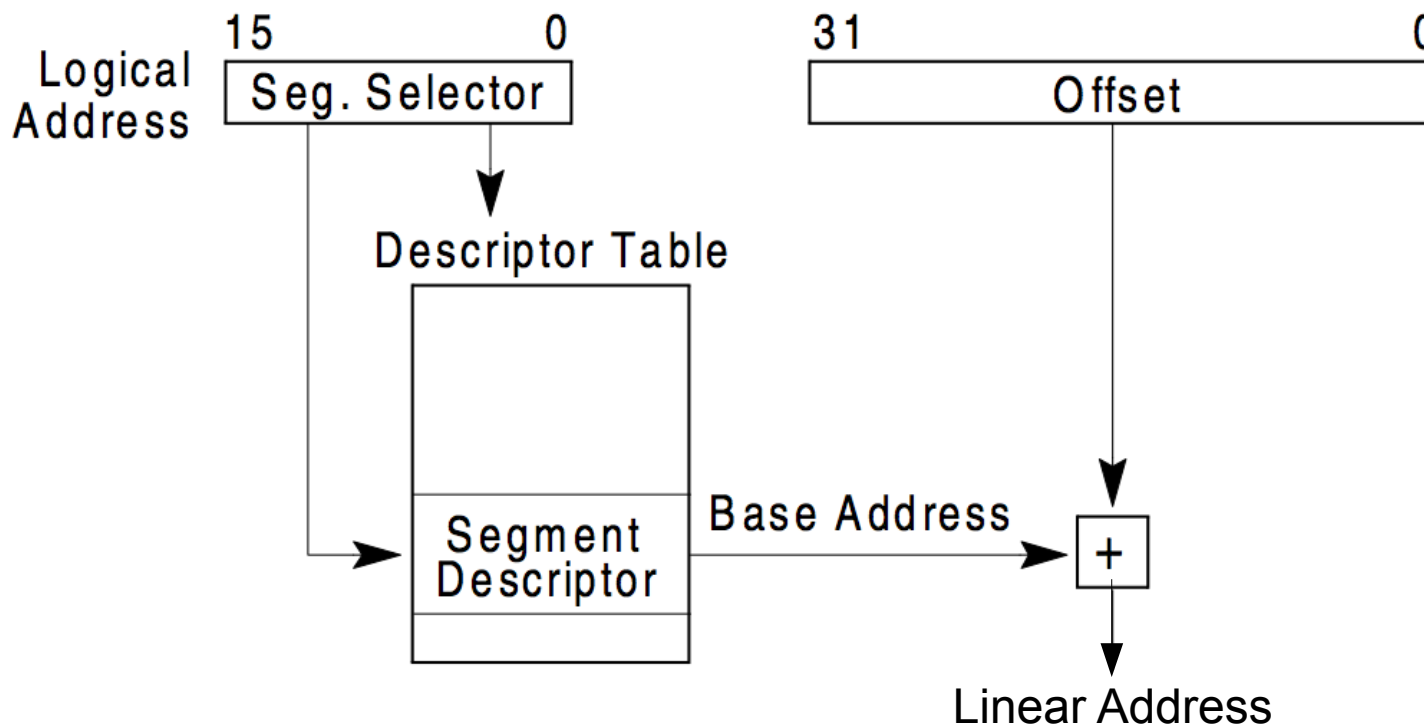
```
{macro definitions}+≡  
#define UMAPD(ptd, frame) \  
    fill_page_table_desc (ptd, true, true, true, frame)  
#define KMAPD(ptd, frame) \  
    fill_page_table_desc (ptd, true, true, false, frame)
```

- für Page Descriptors:

```
{macro definitions}+≡  
#define UMAP(pd, frame)  
    fill_page_desc (pd, true, true, true, false, frame)  
#define KMAP(pd, frame)  
    fill_page_desc (pd, true, true, false, false, frame)
```

# Identity Mapping (1)

- Paging und Segmentierung arbeiten zusammen
  - Schritt 1: **Logische Adresse** (Segment + Offset) in **lineare Adresse** umwandeln, z. B.  
 $0x08:0xC0101234 \rightarrow 0x101234$   
(wegen Base =  $0x40000000$  laut Segm.-Desk.)
  - Schritt 2: **lineare Adresse** in **phys. Adresse** umwandeln, z. B.  
 $0x101234 \rightarrow 0x101234$   
(Identity Mapping: anfangs linear = phys.)
  - später: Base auf 0 setzen und „direkt“ hohe Adressen auf phys. Adressen mappen



Bildquelle: Intel Architecture Software Developer's Manual Volume 3: System Programming Guide, p. 3-7

Bildquelle: Intel Architecture Software Developer's Manual Volume 3: System Programming Guide, p. 3-20

# Identity Mapping (3): Drei Phasen

1) Segmentierung  
(mit Base-0x400...-Trick),  
kein Paging

C012.3456

Segment. + 4000.0000

0012.3456

2) Segmentierung  
(mit Base-0x400...-Trick),  
Paging: Ident. Mapping

C012.3456

Segment. + 4000.0000

0012.3456

Paging Ident. Map.

0012.3456

3) Segmentierung  
(mit Base = 0),  
Paging: normal

C012.3456

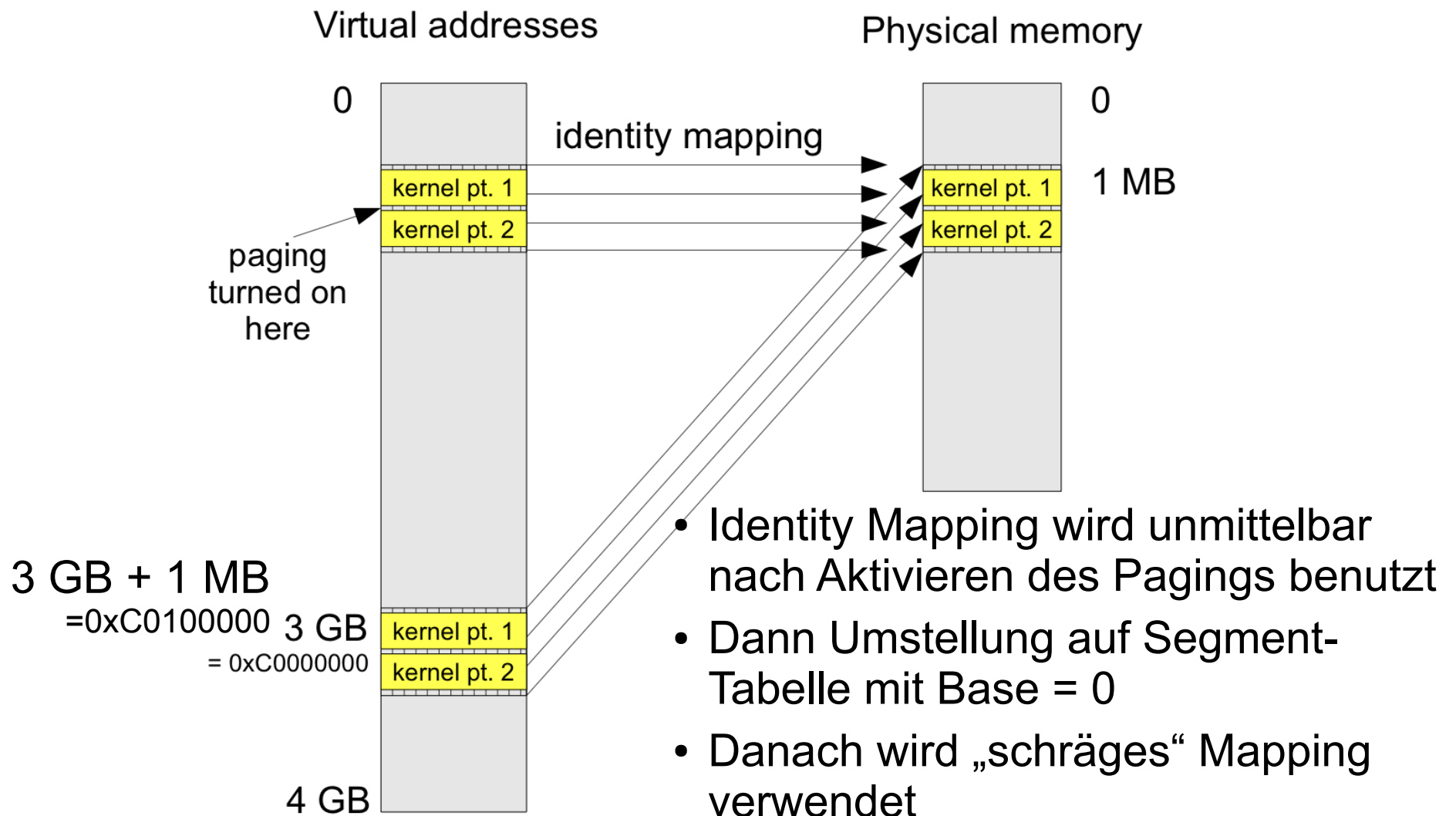
Segment. + 0

C012.3456

Paging

0012.3456

# Identity Mapping (4)



# Identity Mapping (5)

```
{setup identity mapping for kernel}≡  
// file page directory with null entries  
for (int i=1; i<1024; i++) {  
    // Note: loop starts with i=1, not i=0  
    fill_page_table_desc ( &(current_pd->ptds[i]),  
                           false, false, false, 0 );  
};  
  
// make page table kernel_pt first entry of page directory  
KMAPD ( &(current_pd->ptds[0]),  
        (unsigned int)(current_pt)-0xC0000000 );  
  
// make page table kernel_pt also 768th entry of page directory  
KMAPD ( &(current_pd->ptds[768]),  
        (unsigned int)(current_pt)-0xC0000000 );  
  
for (int i=0; i<1024; i++) {  
    KMAP ( &(current_pt->pds[i]), i*4096 );  
}
```

# Paging einschalten

```
<enable paging for the kernel> ≡  
unsigned int cr0;  
  
// Write page directory address  
char *kernel_pd_address;  
kernel_pd_address = (char*)(current_pd) - 0xC0000000;  
asm ("mov %0, %%cr3" : : "r"(kernel_pd_address)); // write CR3  
  
// Enable paging by setting PG bit 31 of CR0  
asm ("mov %%cr0, %0" : "=r"(cr0) : ); // read CR0  
cr0 = cr0 | (1<<31);  
asm ("mov %0, %%cr0" : : "r"(cr0) ); // write CR0
```

# gdt\_install() (1)

- Nach dem Aktivieren des Paging:  
neue GDT erzeugen und laden

```
< type definitions > +≡  
struct gdt_ptr {  
    unsigned int limit : 16;  
    unsigned int base  : 32;  
} __attribute__((packed));
```

```
< type definitions > +≡  
struct gdt_entry {  
    unsigned int limit_low    : 16;  
    unsigned int base_low    : 16;  
    unsigned int base_middle : 8;  
    unsigned int access      : 8;  
    unsigned int flags       : 4;  
    unsigned int limit_high  : 4;  
    unsigned int base_high   : 8;  
};
```



# `gdt_install()` (2)

- Einen GDT-Eintrag erzeugen:

```
{function implementations}+≡
void gdt_set_gate (int num, unsigned long base,
    unsigned long limit, unsigned char access,
    unsigned char gran) {
    /* Setup the descriptor base address */
    gdt[num].base_low = (base & 0xFFFF);           // 16 bits
    gdt[num].base_middle = (base >> 16) & 0xFF;   // 8 bits
    gdt[num].base_high = (base >> 24) & 0xFF;     // 8 bits

    /* Setup the descriptor limits */
    gdt[num].limit_low = (limit & 0xFFFF);         // 16 bits
    gdt[num].limit_high = ((limit >> 16) & 0x0F); // 4 bits

    /* Finally, set up the granularity and access flags */
    gdt[num].flags = gran & 0xF;
    gdt[num].access = access;
}
```

# `gdt_install()` (3)

- Neue GDT mit Base = 0 schreiben:

```
{function implementations}+≡
void gdt_install() {
    gp.limit = (sizeof(struct gdt_entry) * 6) - 1;
    gp.base = (int) &gdt;

    // NULL descriptor
    gdt_set_gate(0, 0, 0, 0, 0);

    // Code segment: Base = 0, Limit = 4 GB
    gdt_set_gate(1, 0, 0xFFFFFFFF, 0b10011010, 0b1100);

    // Data segment: Base = 0, Limit = 4 GB
    gdt_set_gate(2, 0, 0xFFFFFFFF, 0b10010010, 0b1100);

    gdt_flush();    // assembler
}
```

vgl.  
Folien  
11 +  
13

# gdt\_install() (4)

```
<start.asm>+≡
extern gp                ; „Pointer“ auf GDT in
                        ; C-Datei deklariert
gdt_flush:
    lgdt [gp]           ; neue GDT laden
    mov ax, 0x10
    mov ds, ax          ; Segmentreg. setzen
    mov es, ax
    mov fs, ax
    mov gs, ax
    mov ss, ax
    jmp 0x08:flush     ; far jump setzt cs
flush:
    ret
```

# Abschlussarbeiten

- Letzte Schritte
  - Identity Mapping wieder löschen
  - Seitentabelle erweitern, um Zugriff auf Video-Speicher (phys: 0xb8000 ...) zu erlauben
  - Seitentabelle erweitern, um direkten Zugriff auf ganzen phys. Speicher zu erlauben  
 $0xD0000000 \dots 0xD3FFFFFF \rightarrow 0 \dots 03FFFFFF$   
( $0x4000000 = 64 \text{ MB}$ )
- Danach ist Speicher vollständig initialisiert