Betriebssystem-Entwicklung mit literater Programmierung

Slide set 5: Booting, Protected Mode, Storage

Winter semester 2015/16

Dr. Hans-Georg Eßer

hgesser@cs.fau.de

http://ohm.hgesser.de/

v1.2, 10/04/2015

(Organizational matters)

- Ulix book is available on website
- From the *next but one* Lecture: read parts of the book beforehand in preparation
- Notes on reading always in the previous lecture (and on the website)

Boat (1)

- General procedure
  - PC executes BIOS code
  - BIOS routine searches for bootable data carriers
  - BIOS loads the boot sector from the data carrier and jumps to the start address of the boot manager
  - Modern boot managers (“second stage boot loaders”) load further code
  - After selection, the boot manager loads the kernel and other files (e.g. initrd) and jumps to the start address of the kernel

Boat (2)

- How we boot ULIX
  - In principle: possible to write your own boot manager
  - easier: use GRUB
  - FAT diskette contains GRUB, the kernel (ulix.bin) and the GRUB configuration (menu.lst)

- *menu.lst* • ≡

```
timeout 5

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

Boat (3)

- Multiboot specification
  - GRUB expects the kernel file to contain a multiboot header (12 bytes) at the beginning:

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

  - Flags: set bits 0 and 1 (load memory aligned, provide memory information to OS)
  - Checksum: - (magic + flags)

Boat (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
```

Storage

- Segmentation (in real mode)
- Segmentation (in protected mode)
- Prepare for paging (virtual memory)
- Paging (→ later)

Segmentation: Real Mode (1)

- When the PC starts, the computer runs in real mode
  → backwards compatible with Intel 8086
- 16-bit register
  → maximum 2_{16} Byte = 64 KByte addressable
- 20-bit addresses possible through segments
  → 2_{20th} Byte = 1 Mbyte addressable
- Segment registers (CS, DS, ...) contain a 16-bit value that is "shifted" four bits to the left
- Access to x + DS << 4 (instead of x)

Segmentation: Real Mode (2)

- Example: [1000: 9abc]

		binary:
DS	= 0x1000	1 0000 0000 0000
DS << 4	= 0x10000	1 0000 0000 0000 0000
adr	= 0x09abc	0 1001 1010 1011 1100
		0 1001 1010 1011 1100 1 0000 0000 0000 0000
Sum: 0x19abc		1 1001 1010 1011 1100

- Possible division of the memory into 16 segments (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)

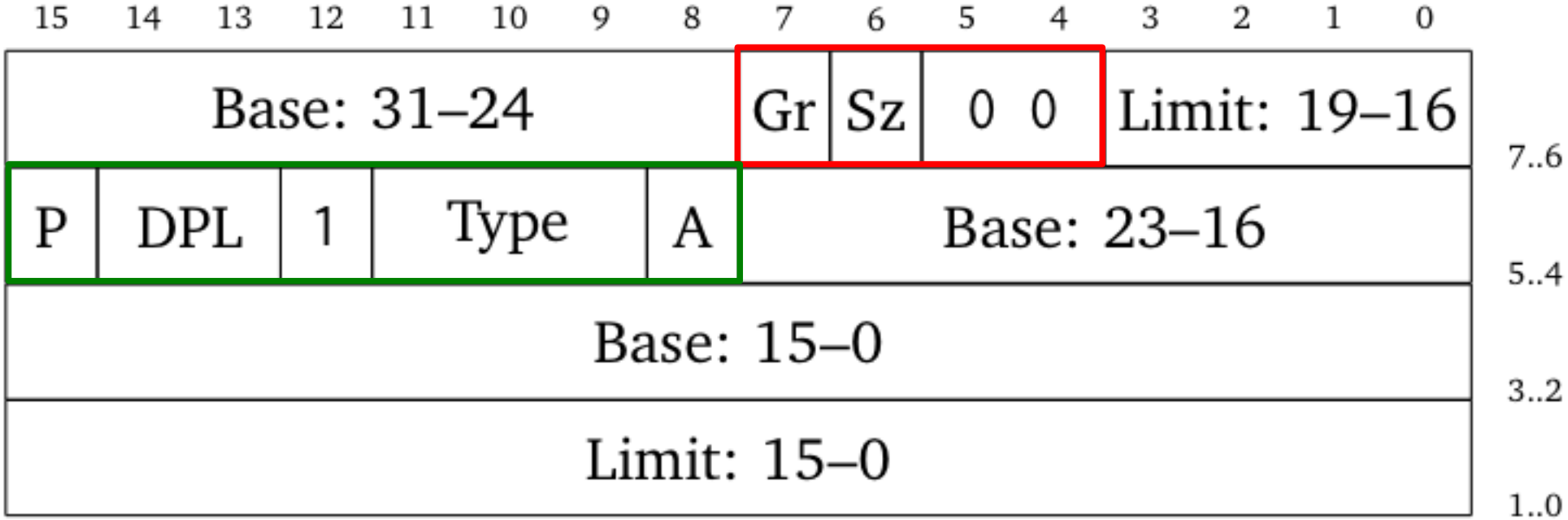
- In protected mode, segmentation is carried out using segment descriptors
- CS, DS etc. do not contain the base address of the segment, but an index in the table of segment descriptors

  → Global Descriptor Table (GDT)

- Each entry is 8 bytes long and contains the values, among other things *Base (* 32-bit) and *Limit (* 20-bit)

  Index always a multiple of 8 (0x08, 0x10, ...)

Segment .: Protected Mode (2)



- Base and limit are not saved “in one piece”
- Flags : 1100 (granularity: 4 KB, 32-bit descr.)
- Access byte : Access rights

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.

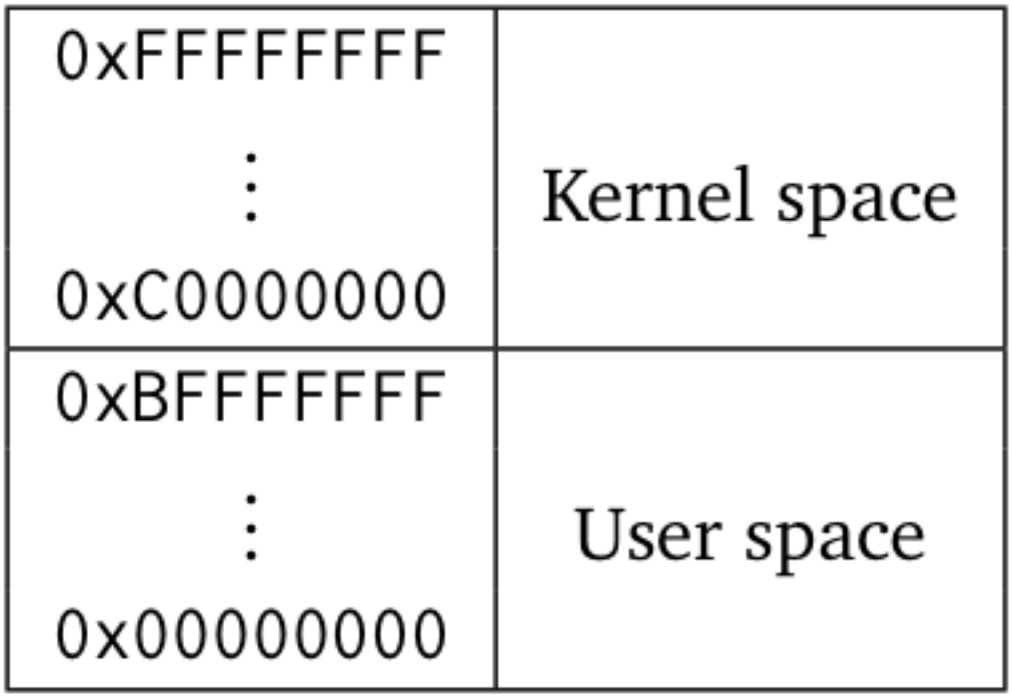
- **0: accessed bit:** we set this to 0; the CPU flips it to 1 when this segment is accessed.

Segm .: Protected Mode (4)

- We need two entries (code, data)
  - 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 ^ ŵa
    (present; ring 0; fixed-1; not executable; grow upwards; allow writing; not accessed).

Preparation for paging (1)

- Desired
  Memory allocation
- Kernel like that
  compile that
  he addresses 0xC0000000
  used
- but where to load?
  → At the beginning paging is not activated



Preparation for paging (2)

- Trick: segment descriptors with base address 0x40000000 produce
- Kernel code from address 0xC0010000 produce
- Example:
- 0xC0010ABC
  + 0x40000000
  = 0x **1** 00010ABC

Carry over (> 32 bit) is omitted

Preparation for paging (3)

- So base: 0x40000000; Limit doesn't matter in principle - we set it up 0xFFFFFFFF

• *start.asm* • + ≡

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

Preparation for paging (4)

Loading the descriptor table:

- *start.asm* • + ≡

```
[section .setup]
begin:
        lgdt [trickgdt]
        mov ax, 0x10
        mov ds, ax
        mov it, ax
        mov fs, ax
        mov gs, ax
        mov ss, ax

        ; far jump
        jmp 0x08: higher half

[section .text]
higher half:
        ; from here in the prot. Fashion
```

Section. set up: from Assembler creates low addresses (from 0x10000) used

(0x08: code, 0x10: data)

Section. text: further Assembler code and C

Kernel - starting with addresses 0xC0100000

Structure of the kernel sources

- seen: start.asm
  - much more does not happen there; only code for interrupt and exception handling is missing
- most of the code is in ulix.c
  - approx. 97% C code
  - partly inline assembler
- there are also two C files
  - printf.c (external implementation of printf)
  - module.c (→ later)

Structure of the C file 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⟩
```

Structure of the C file ulix.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⟩
  }
```

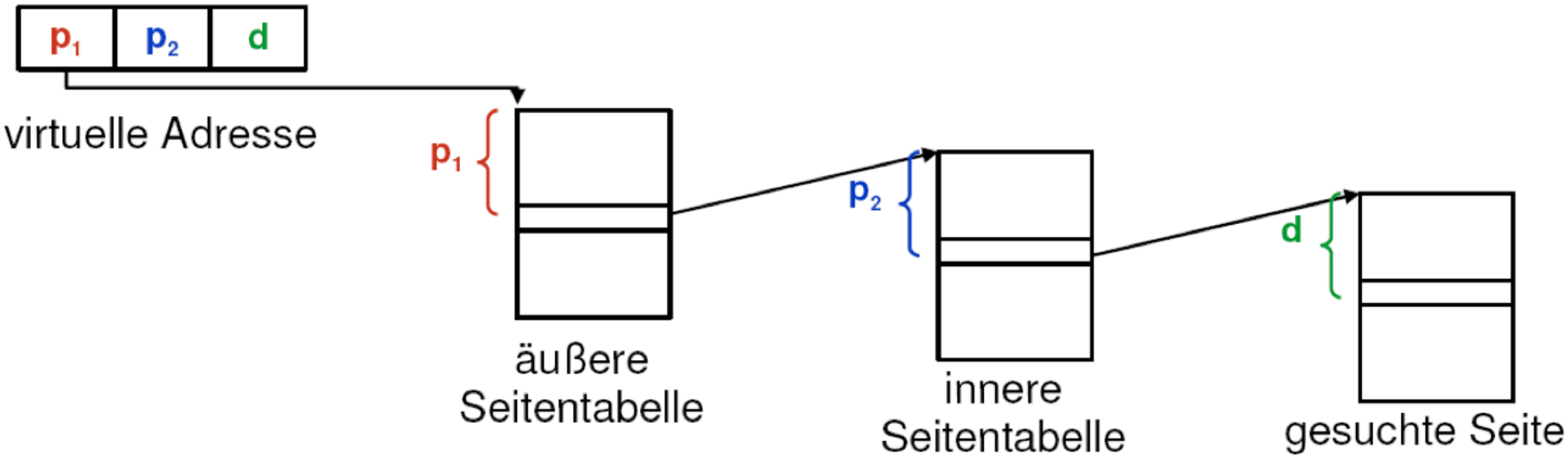
Activate paging

⟨*setup memory* 78b⟩≡

```
⟨setup identity mapping for kernel 161a⟩
⟨enable paging for the kernel 161b⟩
gdt_install();
```

Intel: Side Tables (1)

- generally multi-level side tables
- z. B. two-stage:

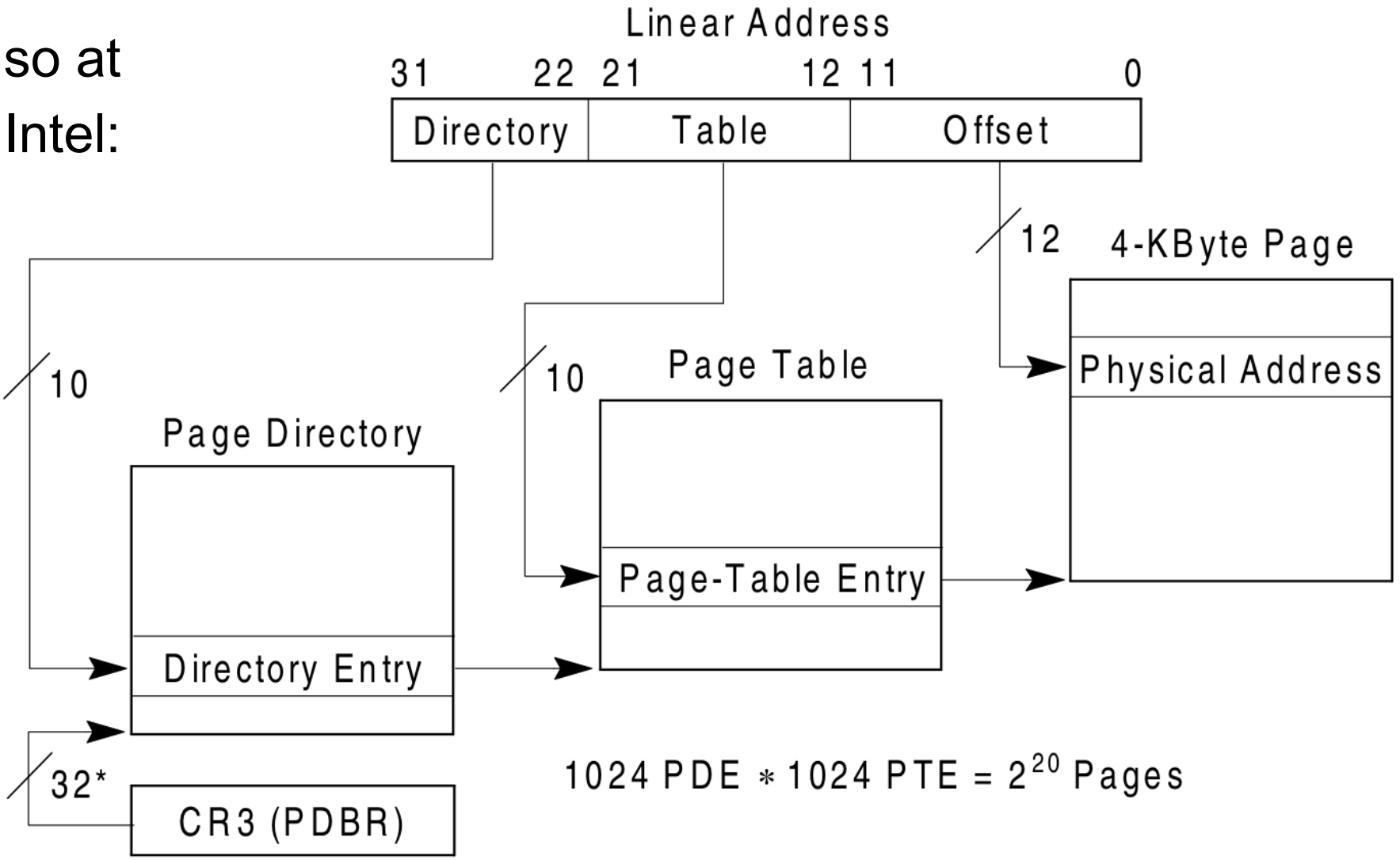


- Intel notation:

  Page Directory (Outside), **Page Table (** Inside)

Intel: Side Tables (2)

- so at Intel:



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

Intel: Side Tables (3)

- An entry in the Page Directory (outside) is called
  - **Page Directory Entry** or
  - **Page table descriptor**
    (points to a page table)
- An entry in the Page Table (inside) is called
  - **Page table entry** or
  - **Page Descriptor (** points to a side frame)
- Structure of the two data types almost the same

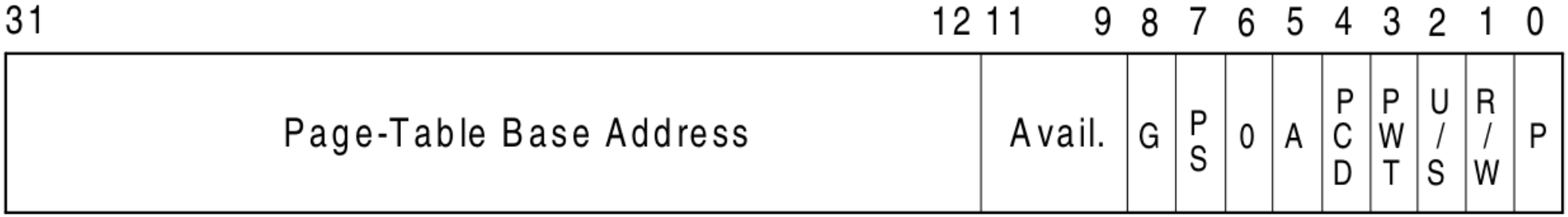
Intel: Side Tables (4)

- Phys. Memory is in **Page frames** divided into 4 KB in size
- Page directories and page tables are also 4 KB in size, so they fit exactly into one frame
- The beginning of a frame (also: a page directory, a page table) is always a multiple of 4 KB
- therefore the top 20 bits of a 32-bit address are sufficient to save the physical address (12 bits → 2^{12} Byte = 4 KB)

ULIX: Side Tables (1)

- Page table descriptor page_table_desc
  - is 32 bits in size
  - contains the upper 20 bits of the physical address of a page table in the upper 20 bits
  - contains attributes in the lower 12 bits
  - Calculate complete physical address = set lower 12 bits to 0

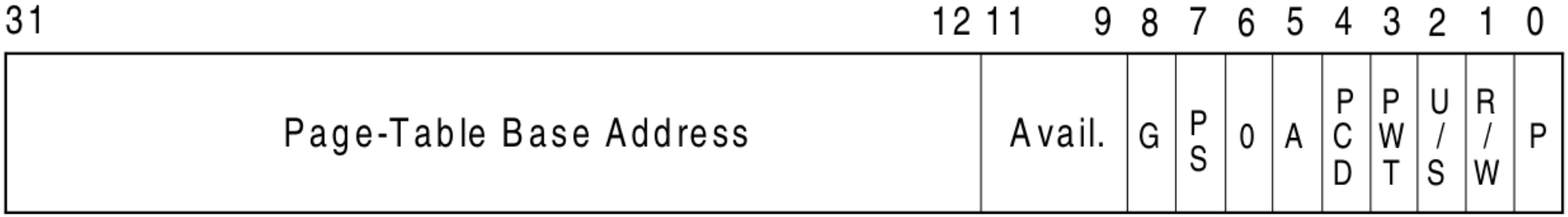
Image source: Intel Architecture Software Developer's Manual Volume 3: System Programming Guide, p. 3-23



ULIX: Side Tables (2)

- *type declarations* • + ≡

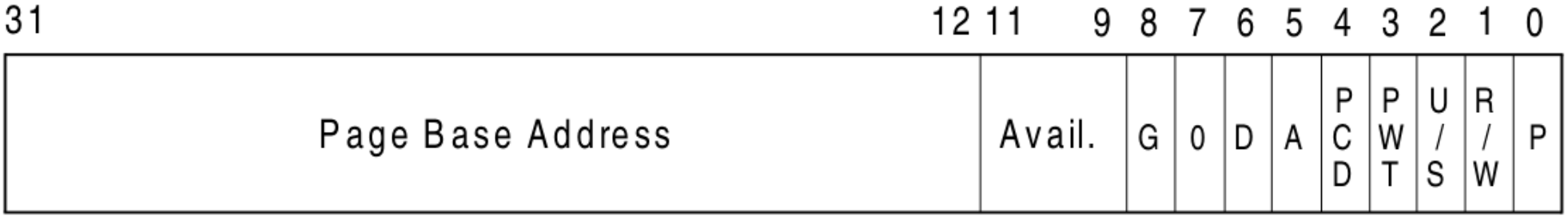
```
typedef struct {
    unsigned int present                          :    1;    //     0
    unsigned int writeable                        :    1;    //     1
    unsigned int user_accessible: unsigned int pwt    1;    //     2
                                                  :    1;    //     3 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



ULIX: Side Tables (3)

- Page descriptor page_desc
  - is 32 bits in size
  - contains the upper 20 bits of the physical address of a page frame in the upper 20 bits
  - contains attributes in the lower 12 bits
  - Calculate complete physical address = set lower 12 bits to 0

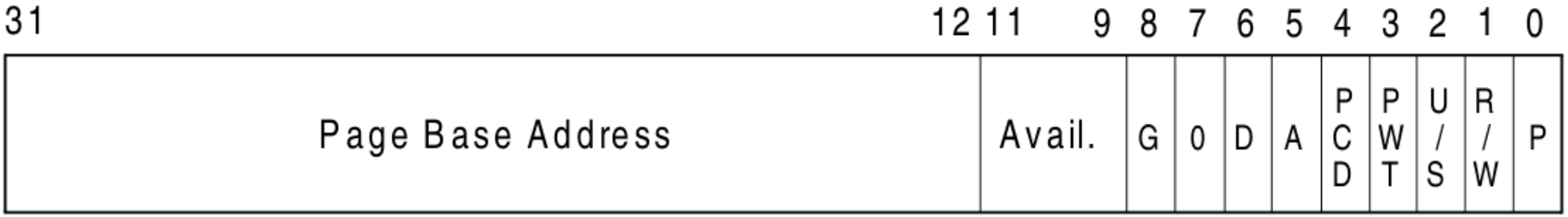
Image source: Intel Architecture Software Developer's Manual Volume 3: System Programming Guide, p. 3-23



ULIX: Side Tables (4)

- *type declarations* • + ≡

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



ULIX: Side Tables (5)

Data structures in comparison

```
typedef struct {
    unsigned int present                    : 1;    //      0
    unsigned int writeable                  : 1;    //      1
    unsigned int user_accessible: 1; unsigned int pwt   //      2
                                            : 1;    //      3
    unsigned int pcd                        : 1;    //      4th
    unsigned int accessed                   : 1;    //      5
    unsigned int undocumented               : 1;    //      6th
    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;
```

ULIX: Side Tables (6)

- Fill the page table descriptor with content:

• *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;
};
```

ULIX: Side Tables (7)

- Fill the page descriptor with content:

• *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: Side Tables (8)

- Macros for easier calling
  - for 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)
```

  - for 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 and segmentation work together
  - Step 1: **Logical address (** Segment + offset) in **linear address** convert, e.g. B.

    0x08: 0xC0101234 → 0x101234

    (because of base = 0x40000000 according to segment descr.)
  - Step 2: **linear address** in **physical address** convert, e.g. B.

    0x101234 → 0x101234
    (Identity mapping: initially linear = phys.)
  - later: set base to 0 and map "direct" high addresses to physical addresses

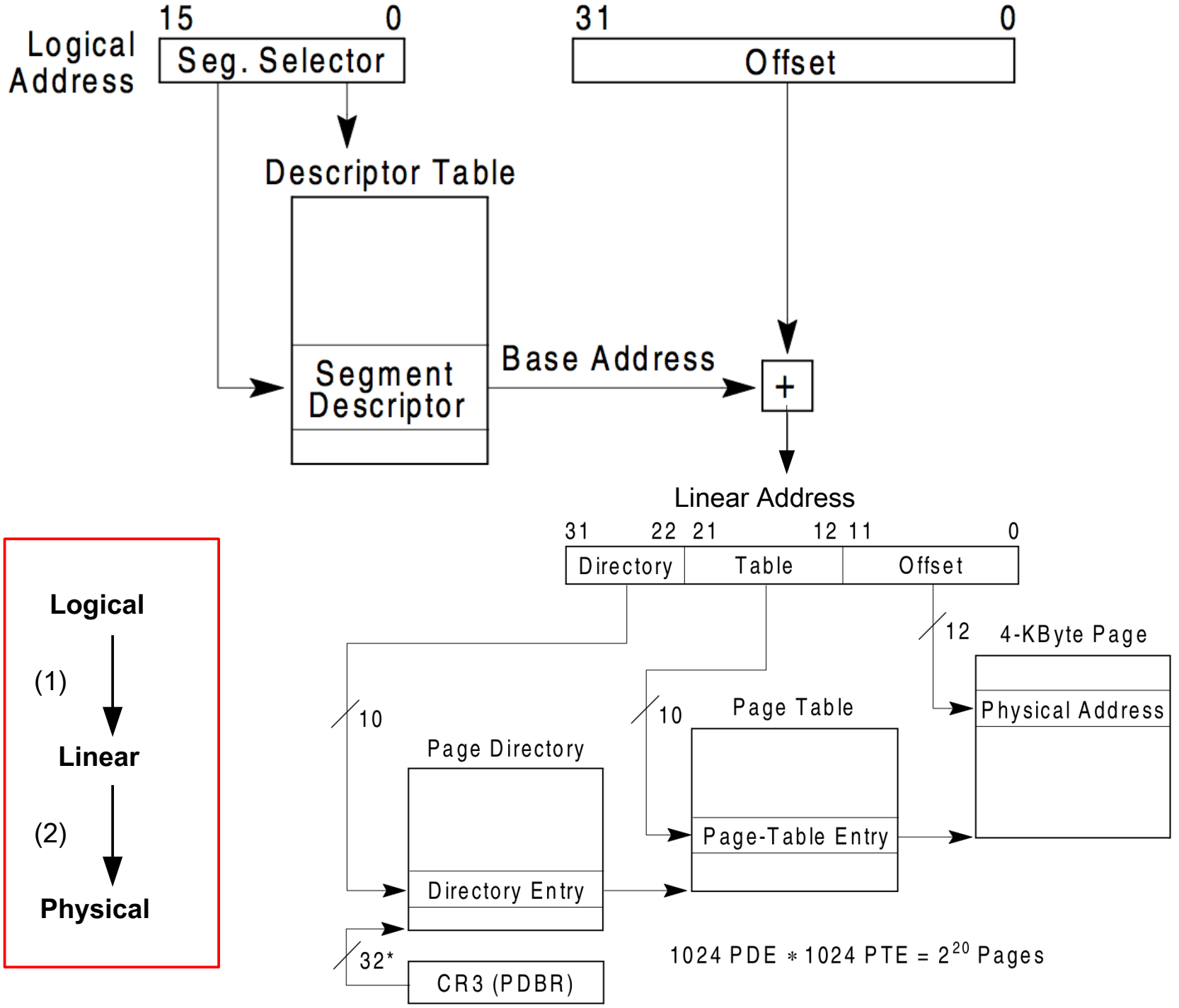


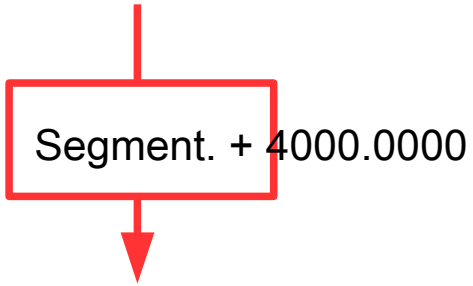
Image source: Intel Architecture Software Developer's Manual Volume 3: System Programming Guide, p. 3-7

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

Identity Mapping (3): Three phases

1) segmentation
(with Base-0x400 ... trick),
no paging

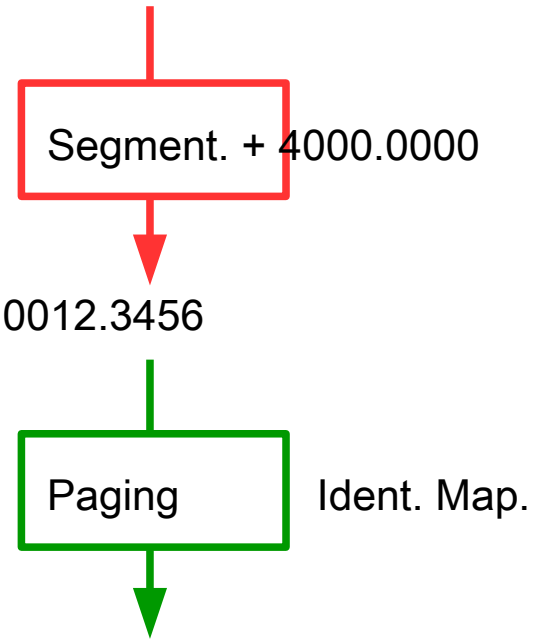
C012.3456



0012.3456

2) segmentation
(with Base-0x400 ... trick),
Paging: Ident. Mapping

C012.3456

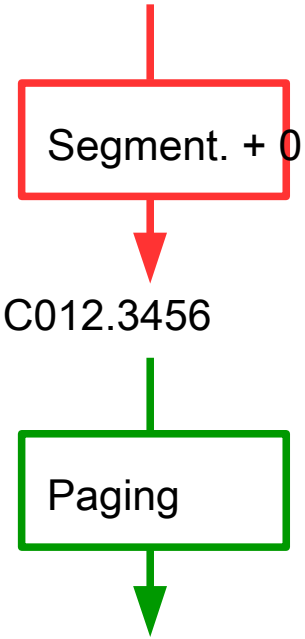


0012.3456

0012.3456

3) segmentation
(with base = 0),
Paging: normal

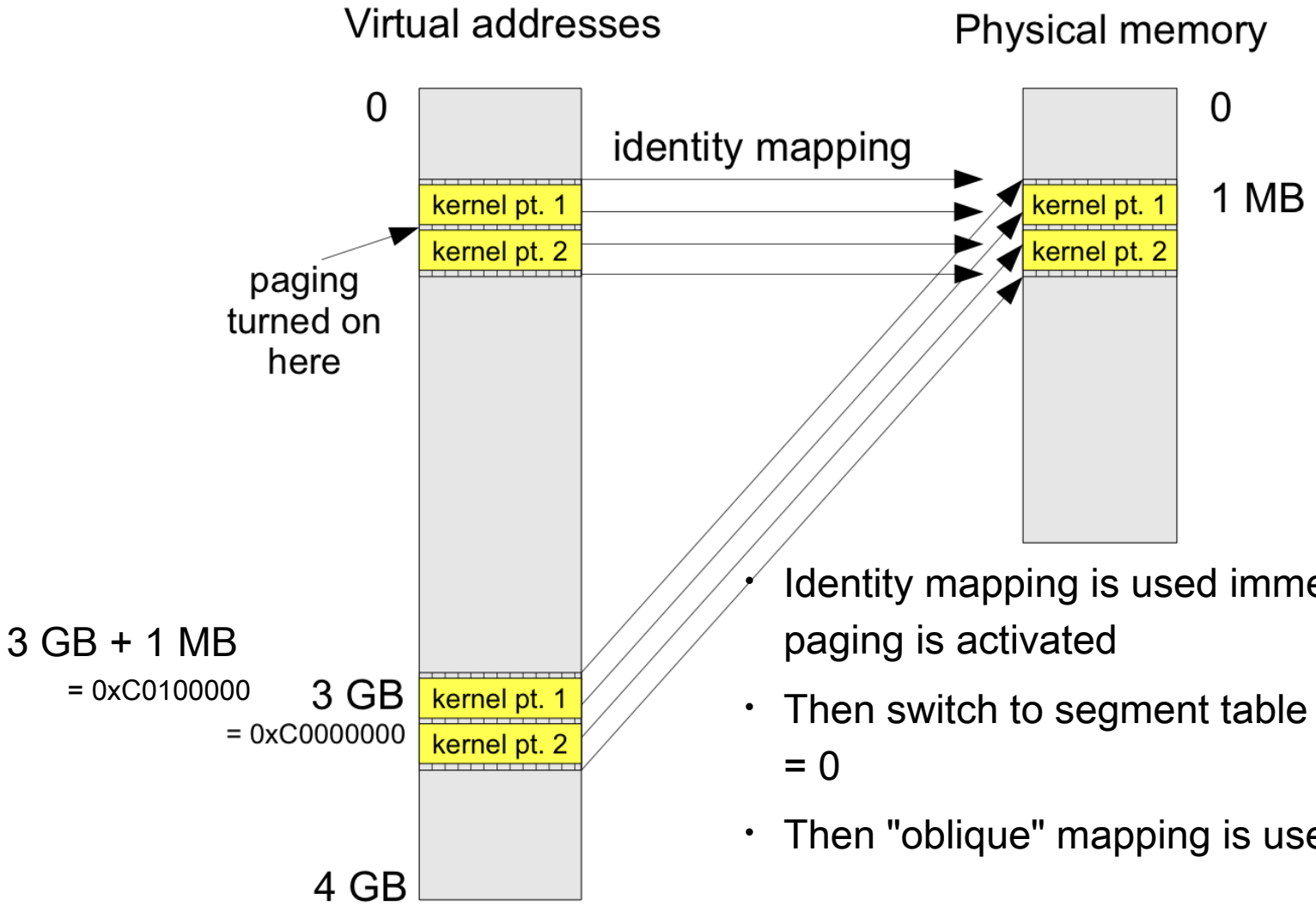
C012.3456



C012.3456

0012.3456

Identity Mapping (4)



- Identity mapping is used immediately after paging is activated
- Then switch to segment table with Base = 0
- Then "oblique" mapping is used

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); }
```

Activate paging

- *enable paging for the kernel* • ≡

```
unsigned int cr0;

// Write page directory address char *
kernel_pd_address;
kernel_pd_address = (char *) (current_pd) - 0xC00000 asm ("mov% 0, %%     00;
cr3":: "r" (kernel_pd_address));                                          // write CR3

// Enable paging by setting PG bit 31 of CR0 asm ("mov %% cr0,%
0": "= r" (cr0):); cr0 = cr0 | (1 << 31);                                 // read CR0

asm ("mov% 0, %% cr0":: "r" (cr0));                                       // write CR0
```

gdt_install () (1)

- After activating paging: create and load new GDT

• *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: unsigned int      8th;
    access                                  : 8th;
    unsigned int flags                      :    4;
    unsigned int limit_high: unsigned int        4;
    base_high                               :    8th;
};
```

gdt_install () (2)

- Create a GDT entry:

• *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); gdt [num] .base_middle = (base >> 16) & 0xFF;          // 16 bits
    gdt [num] .base_high = (base >> 24) & 0xFF;                             // 8 bits
                                                                            // 8 bits

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

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

    gdt [num] .access = access;
}
```

gdt_install () (3)

- Write new GDT with Base = 0:

• *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
}
```

see. Foils 11 + 13

gdt_install () (4)

- *start.asm* • + ≡

```
external gp                    ; "Pointer" to GDT in; C file
                               declared

gdt_flush:
        lgdt [gp]              ; load new GDT
        mov ax, 0x10
        mov ds, ax             ; Segment reg. put
        mov it, ax
        mov fs, ax
        mov gs, ax
        mov ss, ax
        jmp 0x08: flush        ; far jump sets cs
flush:
        ret
```

Theses

- Final steps
  - Delete identity mapping again
  - Expand page table to access video memory (phys: 0xb8000 …) to allow
  - Extend the page table to allow direct access to the entire physical memory

    0xD0000000 ... 0xD3FFFFFF → 0 ... 03FFFFFF
    (0x4000000 = 64 MB)
- The memory is then completely initialized