Chapter 2. Memory Addressing

- Today's microprocessors include several circuits to make memory managment both more efficient and more robust
- In this chapter we study details on how 80x86 (IA-32) microprocessors address memory chips and how Linux uses the available addressing circuits. At the same time...
  - You will better understand the theory of paging
  - You will learn how to research the implementation on other platforms
- This is the first of three chapters related to memory management
  - Chapter 8 discusses how the kernel allocates main memory to itself
  - Chapter 9 considers how linear addresses are assigned to processes

2.1. Memory Addresses

- Programmers refer to a memory address as the way to access a memory cell. But when dealing with 80 x 86 microprocessors, we have to distinguish three kinds of addresses:
  - **Logical address:** Included in the machine language instructions to specify the address of an operand or of an instruction.
    - embodies the well-known 80 x 86 segmented architecture.
    - consists of a segment and an offset
  - **Linear address (also known as virtual address):** A single 32-bit unsigned integer
    - can be used to address up to 4 GB
    - usually represented in hexadecimal notation (from 0x00000000 to 0xffffffff)
  - **Physical address:** Used to address memory cells in memory chips.
    - They correspond to the electrical signals sent along the address pins of the microprocessor to the memory bus.
    - Physical addresses are represented as 32-bit or 36-bit unsigned integers.

Memory Managment Unit (MMU)

- Translation
- Protection

Figure 2-1. Logical address translation

2.2. Segmentation in Hardware

- Starting with the 80286 model, Intel microprocessors perform address translation in two different ways called **real mode and protected mode.**

- We'll focus in the next sections on address translation when **protected mode is enabled**.

- Real mode exists mostly to maintain processor **compatibility with older models** and to **allow the operating system to bootstrap**.

2.2.1. Segment Selectors and Segmentation Registers

- **A logical address consists of two parts**: a segment identifier and an offset that specifies the relative address within the segment.

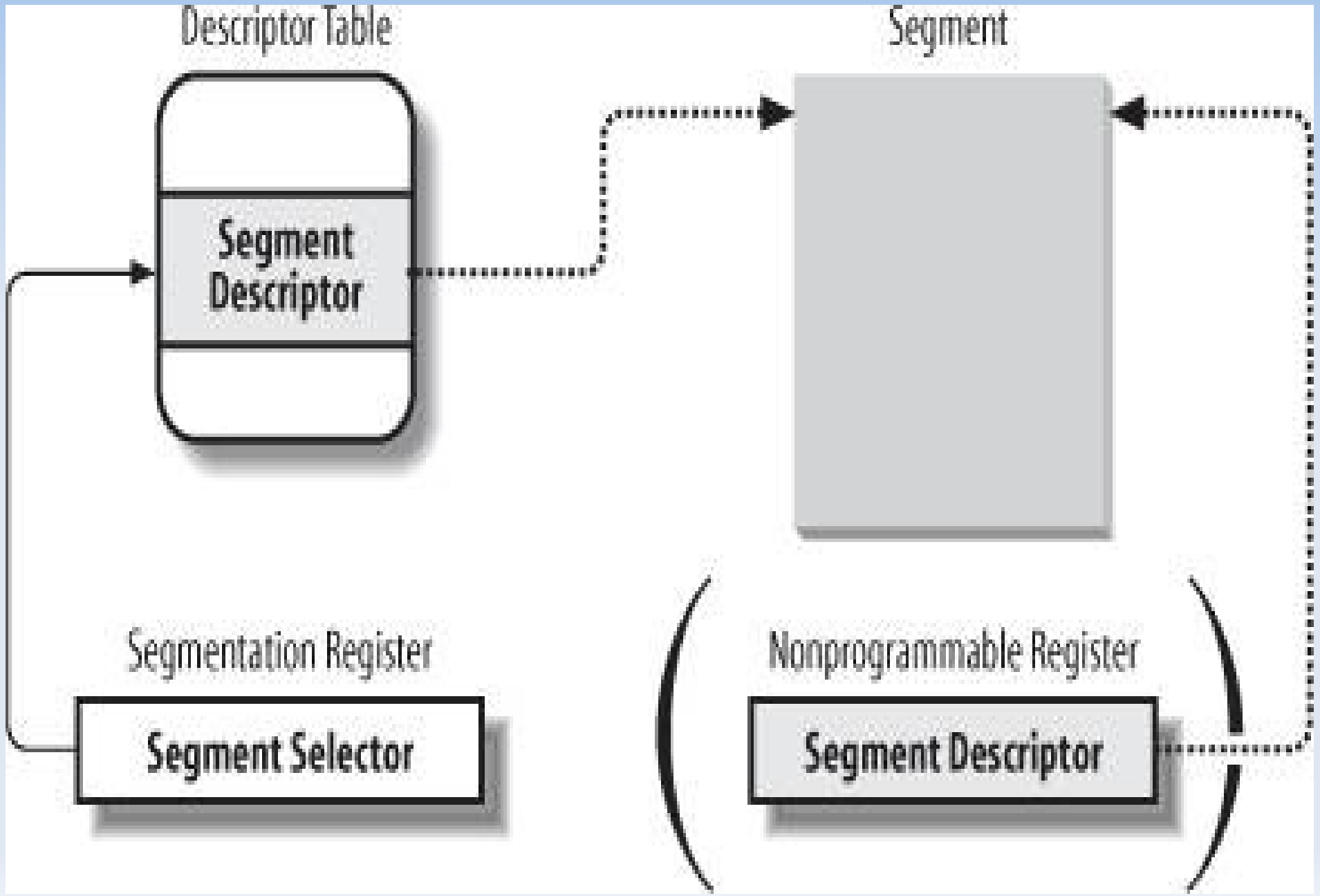
- The segment identifier is a 16-bit field called the **Segment Selector**, while the offset is a 32-bit field.

- To make it easy to retrieve segment selectors quickly, the processor provides **segmentation registers** whose only purpose is to hold Segment Selectors; these registers are called cs, ss, ds, es, fs, and gs.

Segmentation registers

- Three of the six segmentation registers have specific purposes:
  - **cs:** The code segment register, which points to a segment containing program instructions
  - **ss:** The stack segment register, which points to a segment containing the current program stack
  - **ds:** The data segment register, which points to a segment containing global and static data
- The **cs register** has another important function: it includes a 2-bit field that specifies the **Current Privilege Level (CPL) of the CPU**. The value 0 denotes the highest privilege level, while the value 3 denotes the lowest one.
  - Linux uses only levels 0 and 3, which are respectively called Kernel Mode and User Mode.

Selector and descriptor

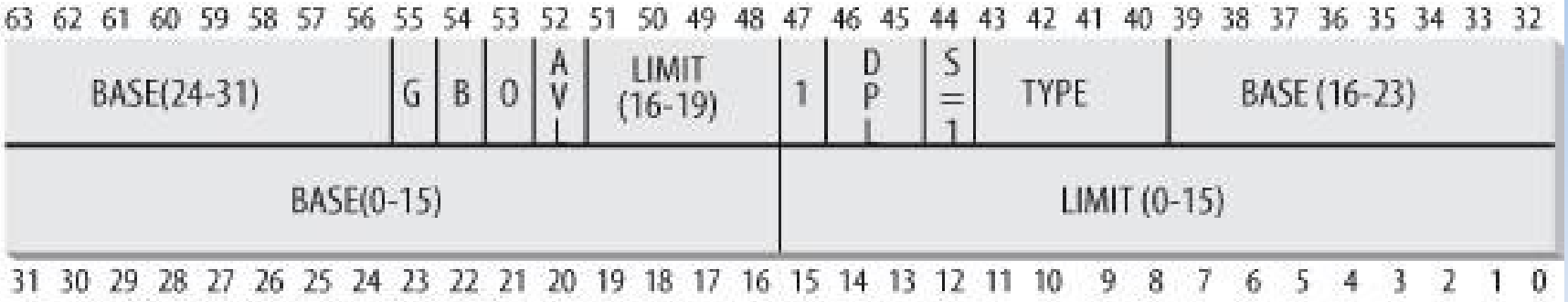


2.2.2. Segment Descriptors

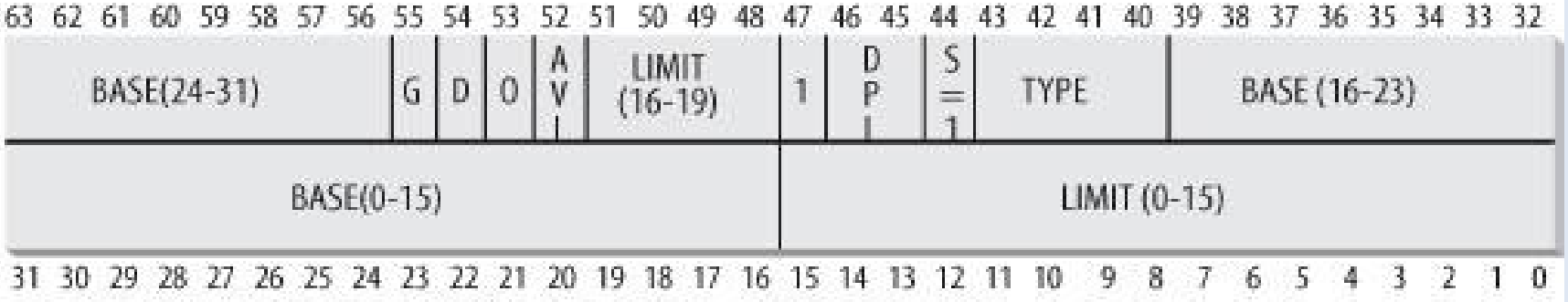
- Each segment is represented by an 8-byte Segment Descriptor that describes the segment characteristics.
- Segment Descriptors are stored either in the **Global Descriptor Table (GDT)** or in the **Local Descriptor Table(LDT)**.
- Usually only **one GDT is defined**, while each process is permitted to have its own LDT if it needs to create additional segments besides those stored in the GDT.
- The address and size of the GDT in main memory are contained in the **gdtr control register**, while the address and size of the currently used LDT are contained in the **ldtr control register**.
- Figure 2-3 illustrates the format of a Segment Descriptor; the meaning of the various fields is explained in Table 2-1.

Segment descriptor format

Data Segment Descriptor



Code Segment Descriptor



System Segment Descriptor

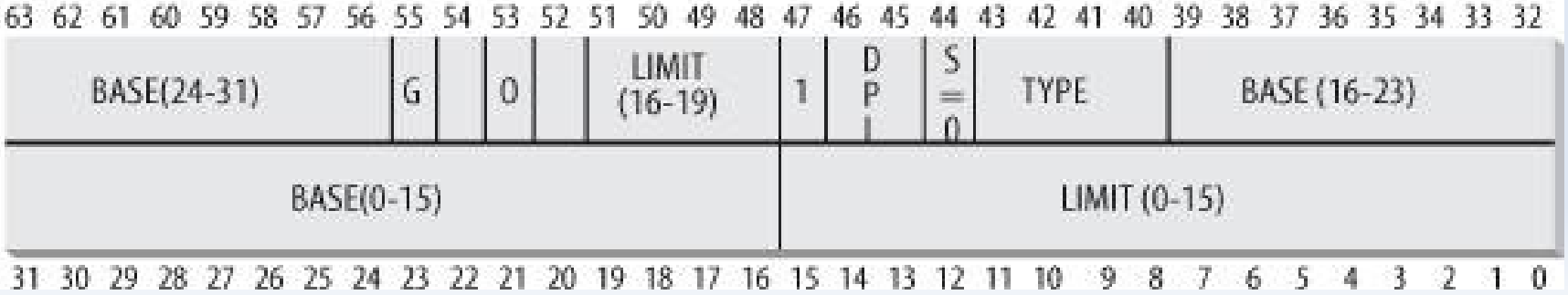


Table 2-1. Segment Descriptor field

- **Base:** Contains the linear address of the first byte of the segment.
- **G:** Granularity flag: if it is cleared (equal to 0), the segment size is expressed in bytes; otherwise, it is expressed in multiples of 4096 bytes.
- **Limit:** Holds the offset of the last memory cell in the segment, thus binding the segment length. When G is set to 0, the size of a segment may vary between 1 byte and 1 MB; otherwise, it may vary between 4 KB and 4 GB.
- **S:** System flag: if it is cleared, the segment is a system segment that stores critical data structures such as the Local Descriptor Table; otherwise, it is a normal code or data segment.
- **Type:** Characterizes the segment type and its access rights.
- **DPL:** Descriptor Privilege Level: used to restrict accesses to the segment. It represents the minimal CPU privilege level requested for accessing the segment. Therefore, a segment with its DPL set to 0 is accessible only when the CPL is 0 that is, in Kernel Mode while a segment with its DPL set to 3 is accessible with every CPL value.
- **P:** Segment-Present flag: is equal to 0 if the segment is not stored currently in main memory. Linux always sets this flag (bit 47) to 1, because it never swaps out whole segments to disk.
- **D or B:** Called D or B depending on whether the segment contains code or data. Its meaning is slightly different in the two cases, but it is basically set (equal to 1) if the addresses used as segment offsets are 32 bits long, and it is cleared if they are 16 bits long (see the Intel manual for further details).
- **AVL:** May be used by the operating system, but it is ignored by Linux.

Types of segments widely used in Linux

- **Code Segment Descriptor:** Indicates that the Segment Descriptor refers to a code segment; it may be included either in the GDT or in the LDT. The descriptor has the S flag set (non-system segment).

- **Data Segment Descriptor:** Indicates that the Segment Descriptor refers to a data segment; it may be included either in the GDT or in the LDT. The descriptor has the S flag set. Stack segments are implemented by means of generic data segments.

- **Task State Segment Descriptor (TSSD):** Indicates that the Segment Descriptor refers to a Task State Segment (TSS) that is, a segment used to save the contents of the processor registers (see the section "Task State Segment" in Chapter 3); it can appear only in the GDT. The corresponding Type field has the value 11 or 9, depending on whether the corresponding process is currently executing on a CPU. The S flag of such descriptors is set to 0.

- **Local Descriptor Table Descriptor (LDTD):** Indicates that the Segment Descriptor refers to a segment containing an LDT; it can appear only in the GDT. The corresponding Type field has the value 2. The S flag of such descriptors is set to 0. The next section shows how 80 x 86 processors are able to decide whether a segment descriptor is stored in the GDT or in the LDT of the process.

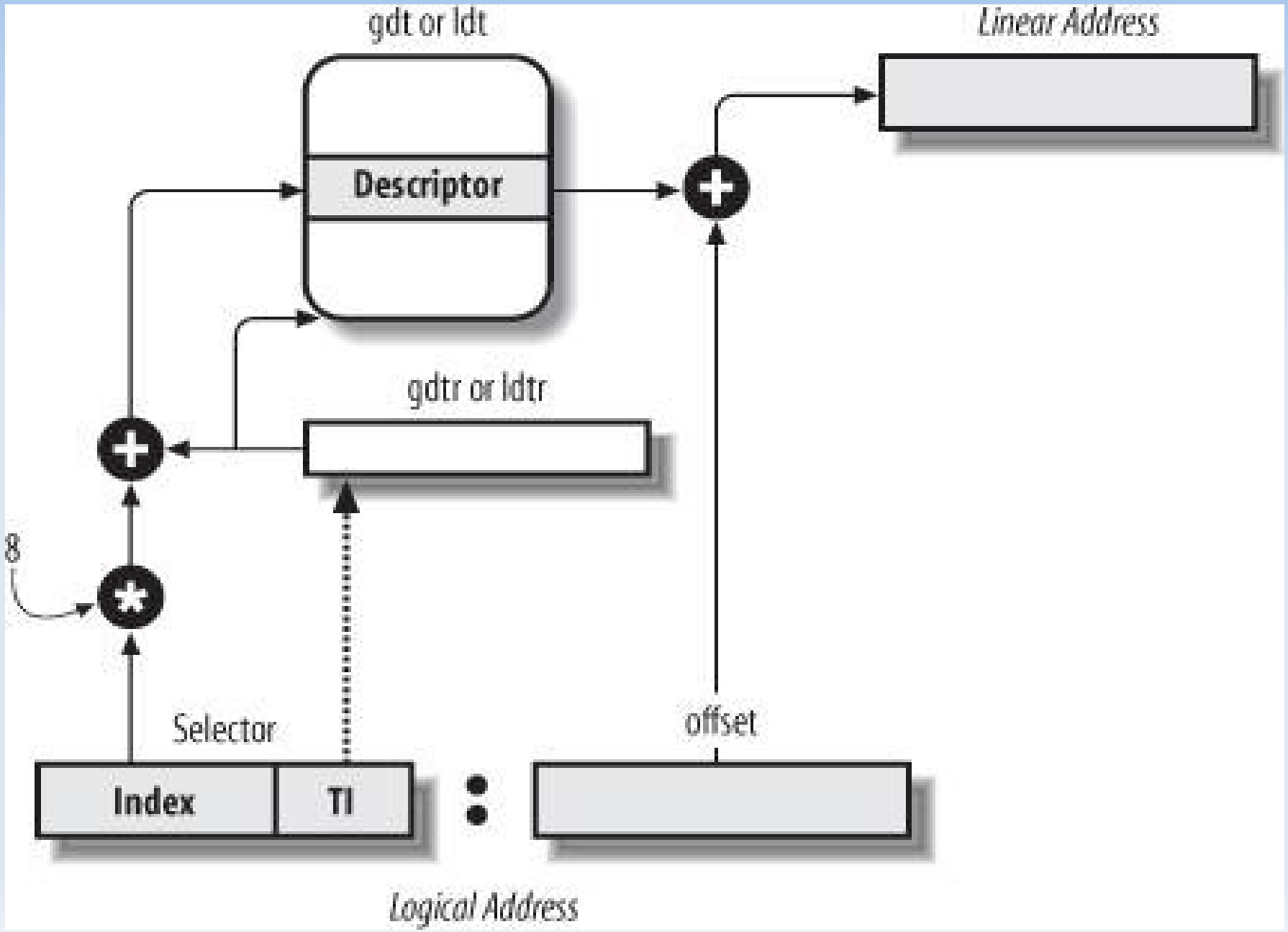
Any Segment Selector includes three fields

- **Index:** Identifies the Segment Descriptor entry contained in the GDT or in the LDT
- **TI:** Table Indicator : specifies whether the Segment Descriptor is included in the GDT (TI = 0) or in the LDT (TI = 1).
- **RPL:** Requestor Privilege Level : specifies the Current Privilege Level of the CPU when the corresponding Segment Selector is loaded into the cs register; it also may be used to selectively weaken the processor privilege level when accessing data segments (see Intel documentation for details).

2.2.4. Segmentation Unit

- Figure 2-5 shows in detail how a logical address is translated into a corresponding linear address. The segmentation unit performs the following operations:
  - Examines the TI field of the Segment Selector to determine which Descriptor Table stores the Segment Descriptor. This field indicates that the Descriptor is either in the GDT (in which case the segmentation unit gets the base linear address of the GDT from the gdtr register) or in the active LDT (in which case the segmentation unit gets the base linear address of that LDT from the ldtr register).
  - Computes the address of the Segment Descriptor from the index field of the Segment Selector. The index field is multiplied by 8 (the size of a Segment Descriptor), and the result is added to the content of the gdtr or ldtr register.
  - Adds the offset of the logical address to the Base field of the Segment Descriptor, thus obtaining the linear address.
- Notice that, thanks to the nonprogrammable registers associated with the segmentation registers, the first two operations need to be performed only when a segmentation register has been changed.

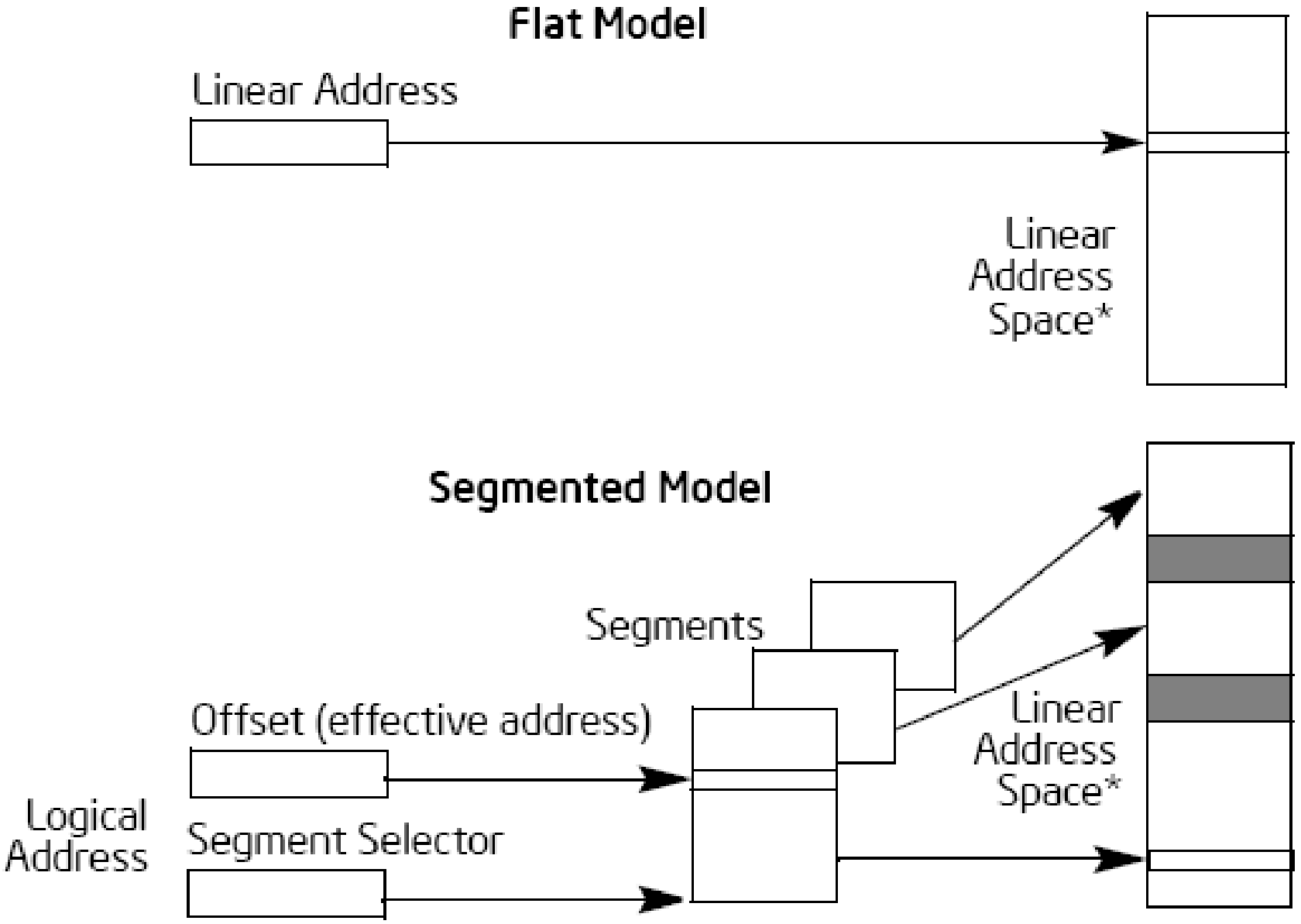
Translating a logical address



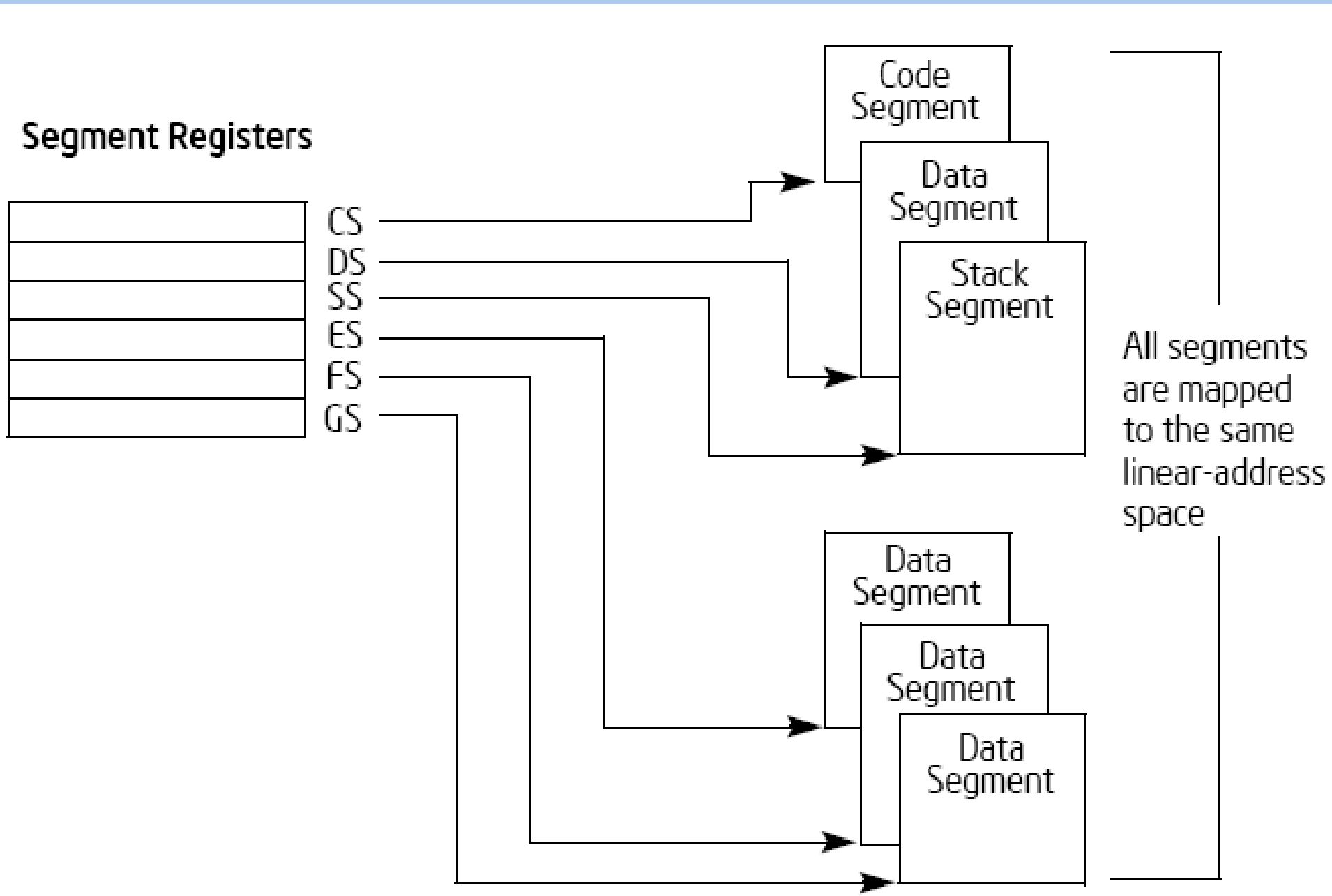
2.3. Segmentation in Linux

- Linux uses segmentation in a very limited way.
- In fact, **segmentation and paging are somewhat redundant**, because both can be used to separate the physical address spaces of processes:
  - segmentation can assign a different linear address space to each process,
  - while paging can map the same linear address space into different physical address spaces.
- **Linux prefers paging to segmentation** for the following reasons:
  - Memory management is simpler when all processes use the same segment register values that is, when they share the same set of linear addresses.
  - One of the design objectives of Linux is portability to a wide range of architectures; RISC architectures in particular have limited support for segmentation.

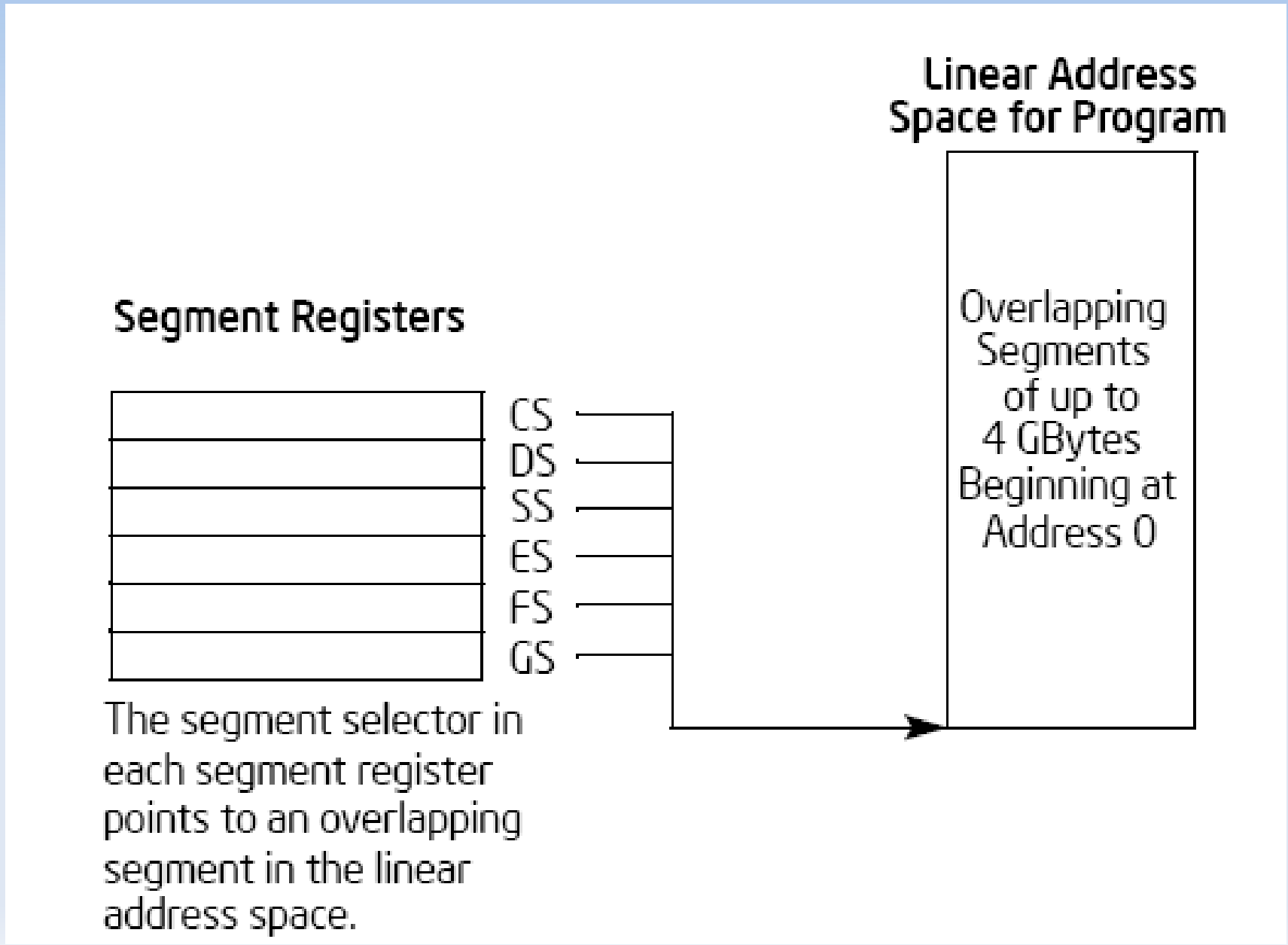
Memory managment models



Use of segment registers in segmented memory model



Use of segment registers in flat memory model



Code and data segments used by Linux

- The 2.6 version of Linux uses segmentation only when required by the 80 x 86 architecture.
- All Linux processes running in User Mode use the same pair of segments to address instructions and data. These segments are called **user code segment and user data segment** , respectively.
- Similarly, all Linux processes running in Kernel Mode use the same pair of segments to address instructions and data: they are called **kernel code segment and kernel data segment** , respectively.
- Table 2-3 shows the values of the Segment Descriptor fields for these four crucial segments.

Segment	Base	G	Limit	S	Type	DPL	D/B	P
user code	0x00000000	1	0xfffff	1	10	3	1	1
user data	0x00000000	1	0xfffff	1	2	3	1	1
kernel code	0x00000000	1	0xfffff	1	10	0	1	1
kernel data	0x00000000	1	0xfffff	1	2	0	1	1

- Besides the four segments just described, Linux makes use of a few other specialized segments. We'll introduce them in the next section while describing the Linux GDT.

Code and data segments used by Linux

- The corresponding Segment Selectors are defined by the macros _ _USER_CS, _ _USER_DS, _ _KERNEL_CS, and _ _KERNEL_DS, respectively. To address the kernel code segment, for instance, the kernel just loads the value yielded by the _ _KERNEL_CS macro into the cs segmentation register.

- Notice that the linear addresses associated with such segments all start at 0 and reach the addressing limit of 2^32 -1. This means that **all processes, either in User Mode or in Kernel Mode, may use the same logical addresses.**

- Another important consequence of having all segments start at 0x00000000 is that in Linux, logical addresses coincide with linear addresses; that is, **the value of the Offset field of a logical address always coincides with the value of the corresponding linear address.**

2.3.1 The Linux GDT

- In uniprocessor systems there is only one GDT, while **in multiprocessor systems there is one GDT for every CPU** in the system.
- All GDTs are stored in the **cpu_gdt_table** array, while the addresses and sizes of the GDTs (used when initializing the gdtr registers) are stored in the **cpu_gdt_descr** array.
- The layout of the GDTs is shown schematically in Figure 2-6. Each GDT includes 18 segment descriptors and 14 null, unused, or reserved entries.
- **Unused entries** are inserted on purpose so that Segment Descriptors usually accessed together are kept in the same 32-byte line of the hardware cache
- All copies of the GDT store identical entries, except for a few cases.
  - First, **each processor has its own TSS segment**, thus the corresponding GDT's entries differ.
  - Moreover, a few entries in the GDT may depend on the process that the CPU is executing (**LDT and TLS Segment Descriptors**).
  - Finally, in some cases a processor may temporarily modify an entry in its copy of the GDT; this happens, for instance, when invoking an APM's BIOS procedure.

Linux GDT

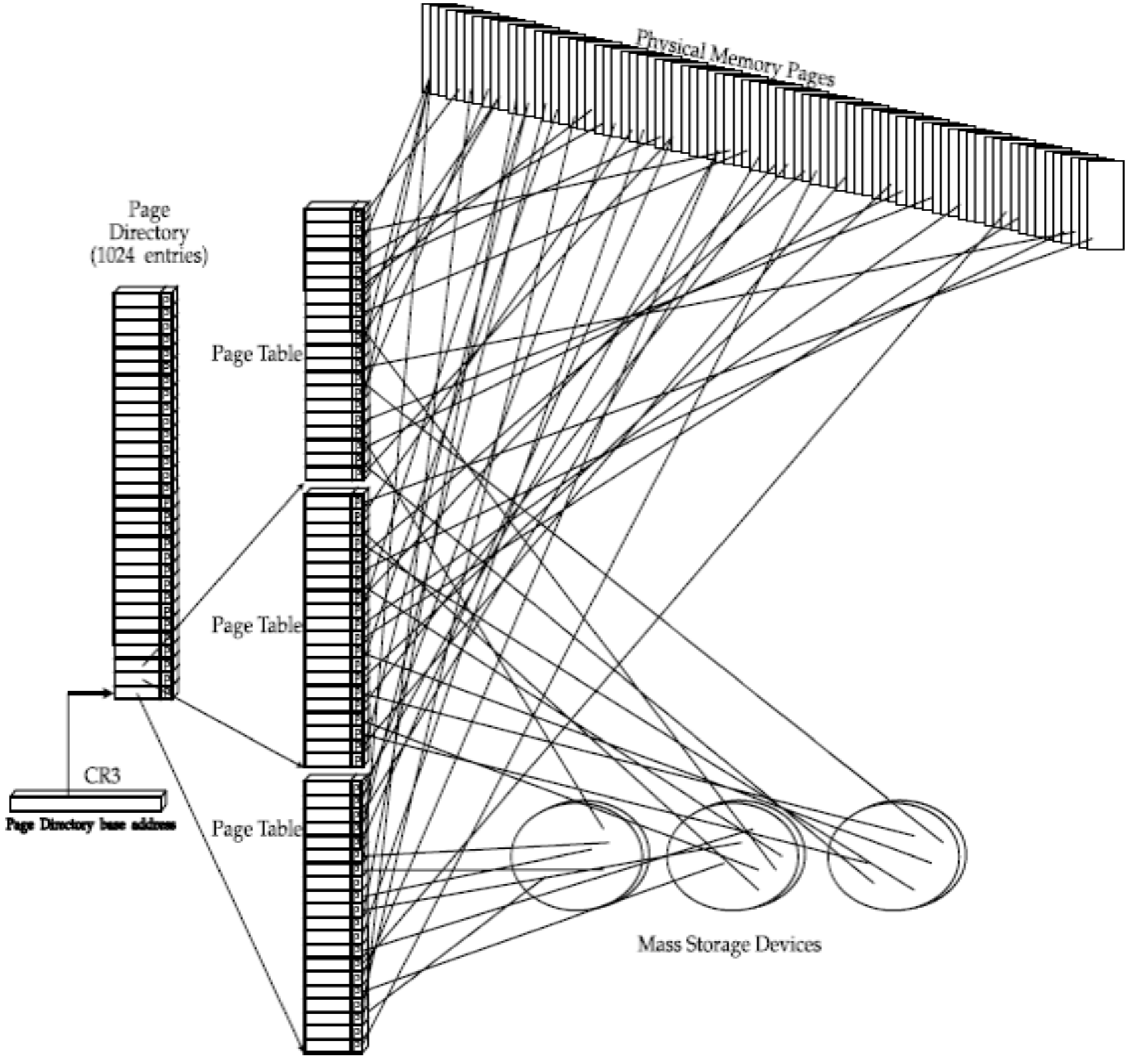
Figure 2-6. The Global Descriptor Table

Linux's GDT	*Segment Selectors*
null	0x0
reserved	
reserved	
reserved	
not used	
not used	
TLS #1	0x33
TLS #2	0x3b
TLS #3	0x43
reserved	
reserved	
reserved	
kernel code	0x60 (__KERNEL_CS)
kernel data	0x68 (__KERNEL_DS)
user code	0x73 (__USER_CS)
user data	0x7b (__USER_DS)

Linux's GDT	*Segment Selectors*
TSS	0x80
LDT	0x88
PNPBIOS 32-bit code	0x90
PNPBIOS 16-bit code	0x98
PNPBIOS 16-bit data	0xa0
PNPBIOS 16-bit data	0xa8
PNPBIOS 16-bit data	0xb0
APMBIOS 32-bit code	0xb8
APMBIOS 16-bit code	0xc0
APMBIOS data	0xc8
not used	
not used	
not used	
not used	
not used	
double fault TSS	0xf8

2.4 Paging in Hardware

- The paging unit translates **linear addresses into physical ones**.
- One key task in the unit is to **check the requested access type against the access rights of the linear address**. If the memory access is not valid, it generates a **Page Fault exception** (see Chapter 4 and Chapter 8).
- For the sake of efficiency, **linear addresses are grouped in fixed-length intervals called pages** ; contiguous linear addresses within a page are mapped into contiguous physical addresses.
- In this way, the kernel can **specify the physical address and the access rights of a page** instead of those of all the linear addresses included in it. Following the usual convention, we shall use the term "page" to refer both to a set of linear addresses and to the data contained in this group of addresses.
- The paging unit thinks of **all RAM as partitioned into fixed-length page frames** (sometimes referred to as physical pages).
  - Each page frame contains a page that is, the length of a page frame coincides with that of a page.
  - A page frame is a constituent of main memory, and hence it is a storage area. **It is important to distinguish a page from a page frame**; the former is just a block of data, which may be stored in any page frame or on disk.


Physical Memory Pages
Page
Directory
(1024 entries)
Page Table
Page Table
Page Table
CR3
Page Directory base address
Mass Storage Devices

Paging in Hardware

- The data structures that map linear to physical addresses are called **page tables** ; they are **stored in main memory** and must be properly initialized by the kernel before enabling the paging unit.

- Starting with the 80386, all 80 x 86 processors support paging; it is enabled by setting the **PG flag of a control register named cr0** .

- **When PG = 0, linear addresses are interpreted as physical addresses.**

2.4.1. Regular Paging

- Starting with the **80386**, the paging unit of Intel processors handles **4 KB pages**.
- The 32 bits of a linear address are divided into three fields:
  - Directory: The most significant 10 bits
  - Table: The intermediate 10 bits
  - Offset: The least significant 12 bits
- The translation of linear addresses is accomplished in two steps, each based on a type of translation table. The first translation table is called the Page Directory, and the second is called the Page Table.
- (In the discussion that follows, the lowercase "page table" term denotes any page storing the mapping between linear and physical addresses, while the capitalized "Page Table" term denotes a page in the last level of page tables.)

multilevel page tables

- If a simple one-level Page Table was used, then it would require up to 2^20 entries (i.e., at 4 bytes per entry, 4 MB of RAM) to represent the Page Table for each process (if the process used a full 4 GB linear address space), even though a process does not use all addresses in that range.

- **The two-level scheme reduces the memory by requiring Page Tables only for those virtual memory regions actually used by a process.**

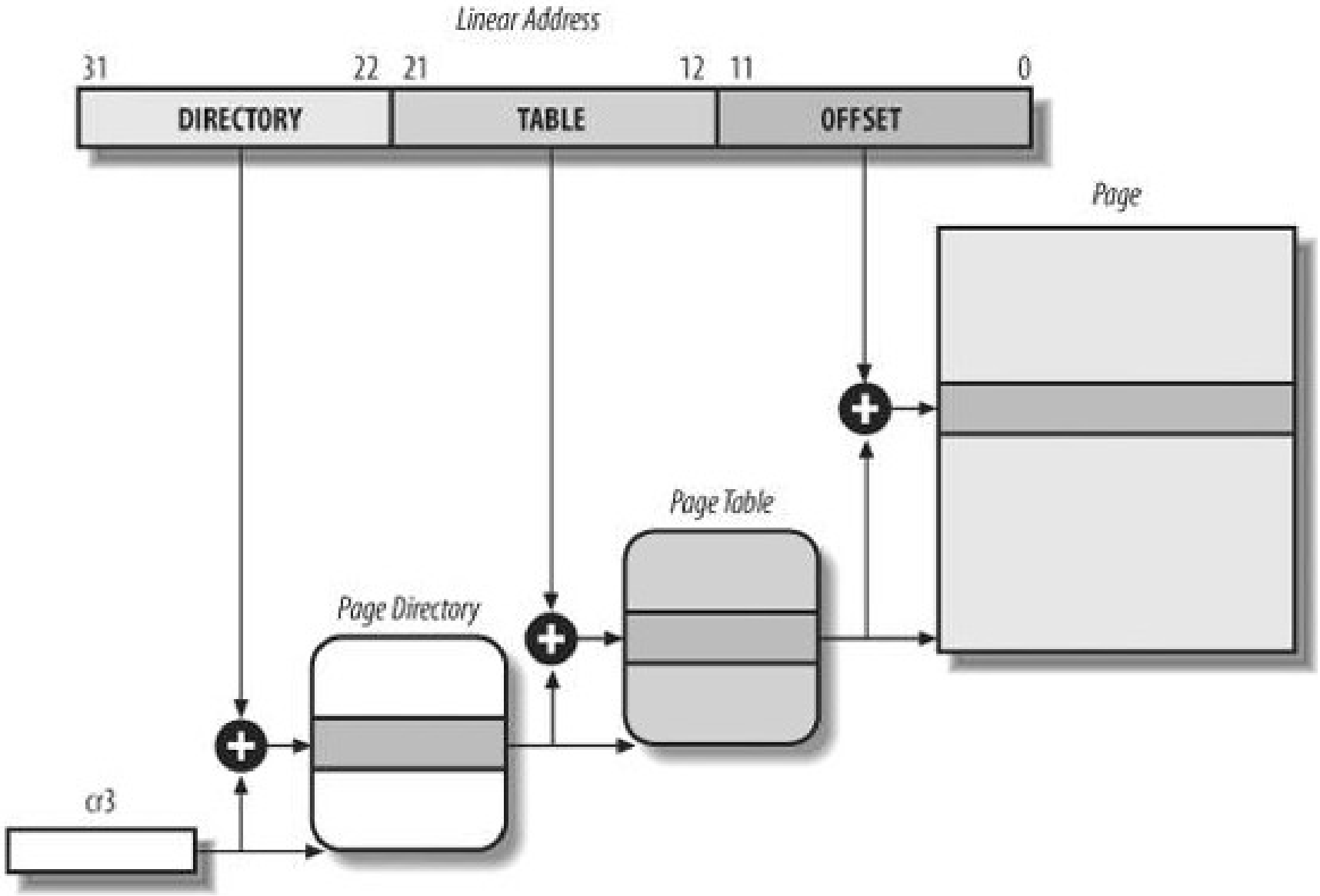
- Each active process must have a Page Directory assigned to it. However, there is no need to allocate RAM for all Page Tables of a process at once; it is more efficient to **allocate RAM for a Page Table only when the process effectively needs it.**

- The physical address of the Page Directory in use is stored in a control register named **cr3** .

- The Directory field within the linear address determines the entry in the Page Directory that points to the proper Page Table. The address's Table field, in turn, determines the entry in the Page Table that contains the physical address of the page frame containing the page. The Offset field determines the relative position within the page frame (see Figure 2-7). Because it is 12 bits long, each page consists of 4096 bytes of data.

2 level paging

Figure 2-7. Paging by 80 x 86 processors



Entry fields

- Both the Directory and the Table fields are 10 bits long, so Page Directories and Page Tables can include up to 1,024 entries. It follows that a Page Directory can address up to 1024 x 1024 x 4096 = 2**32 memory cells, as you'd expect in 32-bit addresses.

- **The entries of Page Directories and Page Tables have the same structure.** Each entry includes the following fields:

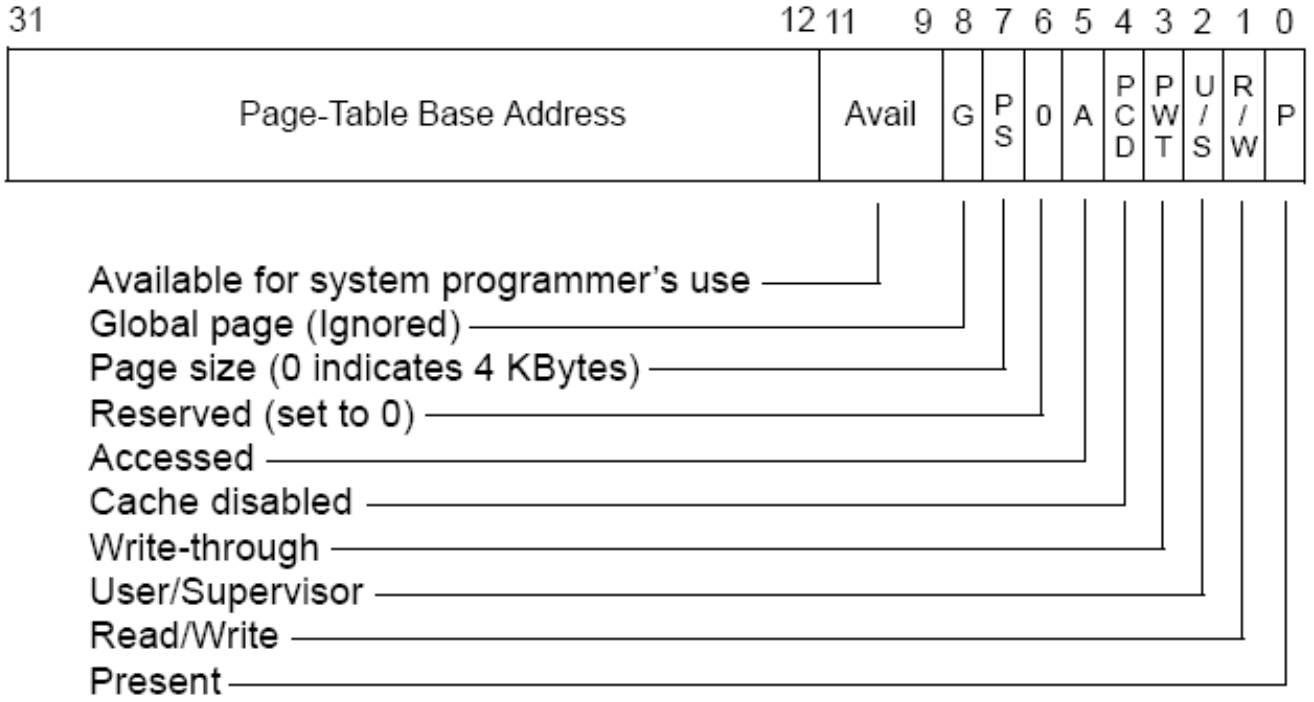
  - **Present flag:** If it is set, the referred-to page (or Page Table) is contained in main memory; if the flag is 0, the page is not contained in main memory and the remaining entry bits may be used by the operating system for its own purposes. If the entry of a Page Table or Page Directory needed to perform an address translation has the Present flag cleared, the paging unit stores the linear address in a control register named cr2 and generates exception 14: the Page Fault exception.

  - **Field containing the 20 most significant bits of a page frame physical address:** Because each page frame has a 4-KB capacity, its physical address must be a multiple of 4096, so the 12 least significant bits of the physical address are always equal to 0. If the field refers to a Page Directory, the page frame contains a Page Table; if it refers to a Page Table, the page frame contains a page of data.

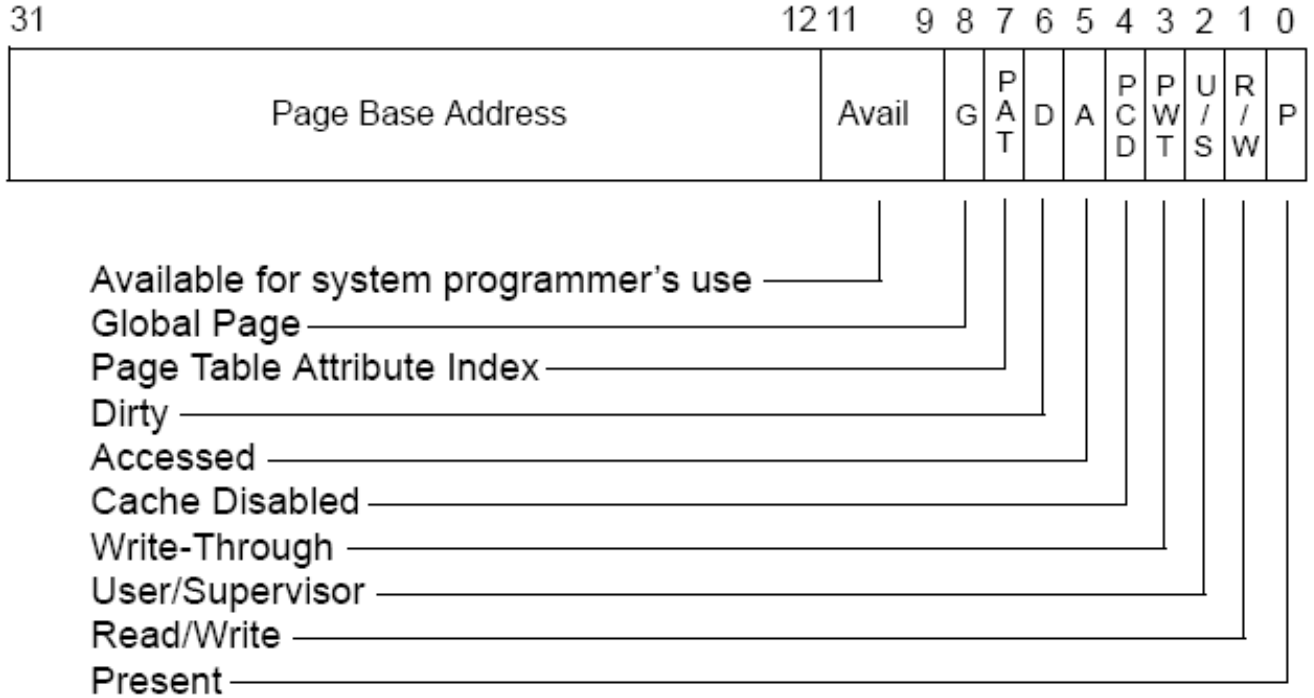
Paging in Hardware

- **Accessed flag:** Set each time the paging unit addresses the corresponding page frame. This flag may be used by the operating system when selecting pages to be swapped out. The paging unit never resets this flag; this must be done by the operating system.
- **Dirty flag:** Applies only to the Page Table entries. It is set each time a write operation is performed on the page frame. As with the Accessed flag, Dirty may be used by the operating system when selecting pages to be swapped out. The paging unit never resets this flag; this must be done by the operating system.
- **Read/Write flag:** Contains the access right (Read/Write or Read) of the page or of the Page Table (see the section "Hardware Protection Scheme" later in this chapter).
- **User/Supervisor flag:** Contains the privilege level required to access the page or Page Table (see the later section "Hardware Protection Scheme").
- **PCD and PWT flags:** Controls the way the page or Page Table is handled by the hardware cache (see the section "Hardware Cache" later in this chapter).
- **Page Size flag:** Applies only to Page Directory entries. If it is set, the entry refers to a 2 MB- or 4 MB-long page frame (see the following sections).
- **Global flag:** Applies only to Page Table entries. This flag was introduced in the Pentium Pro to prevent frequently used pages from being flushed from the TLB cache (see the section "Translation Lookaside Buffers (TLB)" later in this chapter). It works only if the Page Global Enable (PGE) flag of register cr4 is set.

Page-Directory Entry (4-KByte Page Table)



Page-Table Entry (4-KByte Page)

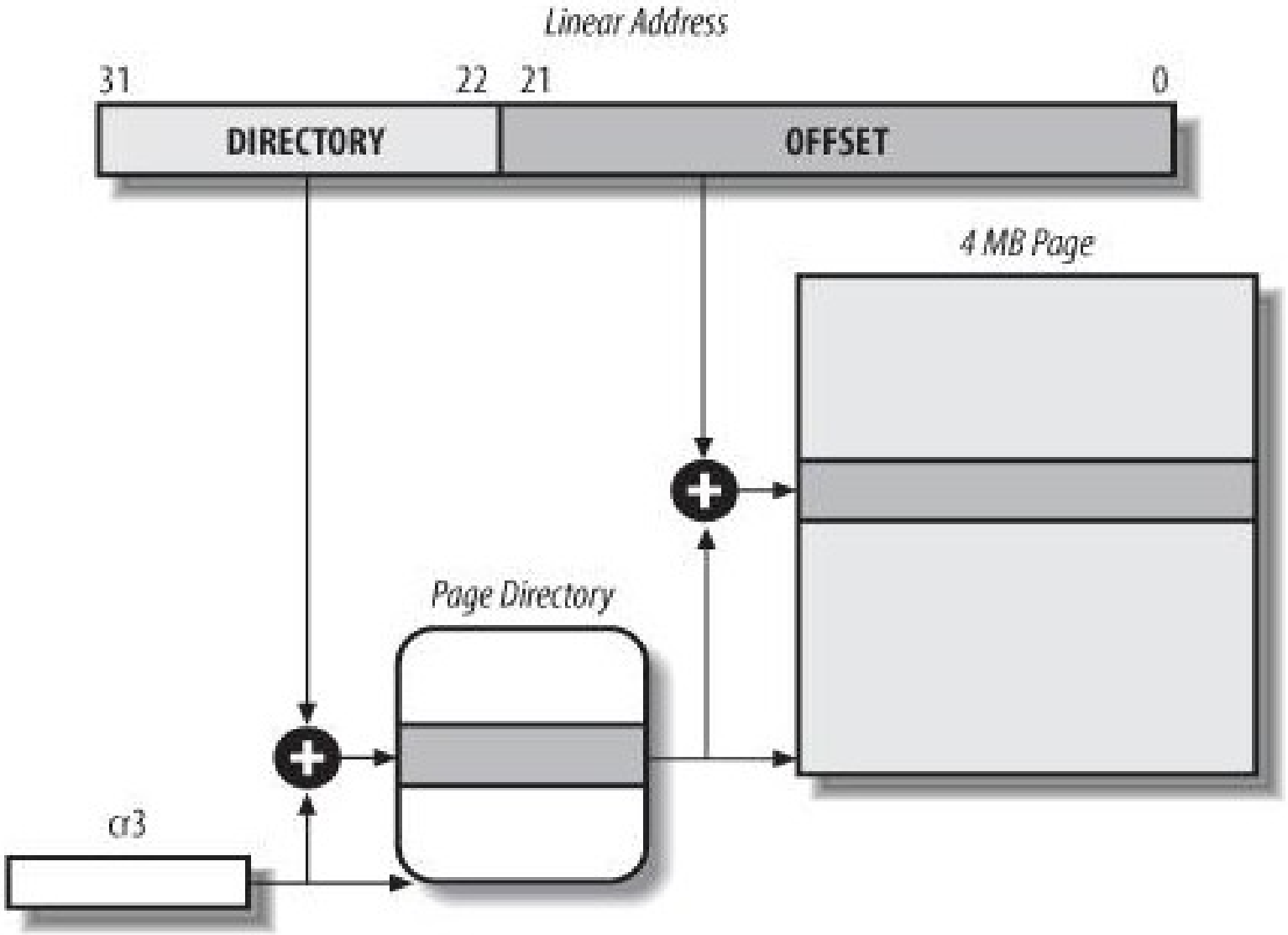


2.4.2. Extended Paging

- Starting with the Pentium model, 80 x 86 microprocessors introduce **extended paging**, which allows **page frames to be 4 MB** instead of 4 KB in size.
- Extended paging is used to **translate large contiguous linear address ranges into corresponding physical ones**; in these cases, the kernel can do without intermediate Page Tables and thus **save memory and preserve TLB entries**.
- As mentioned in the previous section, extended paging is enabled by setting the **Page Size flag of a Page Directory entry**. In this case, the paging unit divides the 32 bits of a linear address into two fields:
  - **Directory**: The most significant 10 bits
  - **Offset**: The remaining 22 bits
- **Page Directory entries for extended paging** are the same as for normal paging, except that:
  - The Page Size flag must be set.
  - Only the 10 most significant bits of the 20-bit physical address field are significant. This is because each physical address is aligned on a 4-MB boundary, so the 22 least significant bits of the address are 0.
- **Extended paging coexists with regular paging; it is enabled by setting the PSE flag of the cr4 processor register.**

Extended paging

Figure 2-8. Extended paging



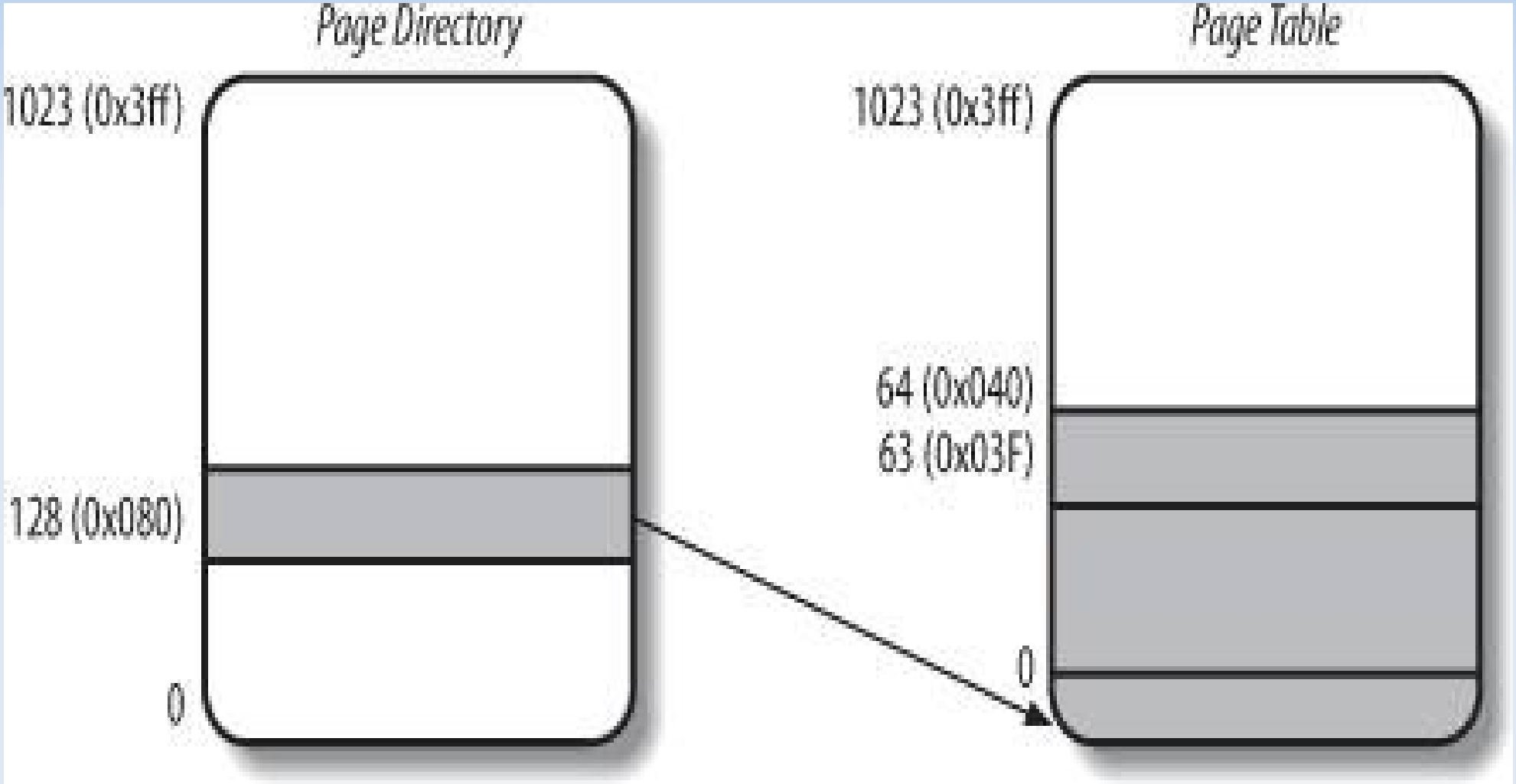
2.4.3. Hardware Protection Scheme

- The paging unit uses a different protection scheme from the segmentation unit.
- While 80 x 86 processors allow four possible privilege levels to a segment, **only two privilege levels are associated with pages and Page Tables**, because privileges are controlled by the **User/Supervisor flag**:
  - When this flag is 0, the page can be addressed only when the CPL is less than 3 (this means, for Linux, when the processor is in Kernel Mode).
  - When the flag is 1, the page can always be addressed.
- Furthermore, instead of the three types of access rights (Read, Write, and Execute) associated with segments, **only two types of access rights (Read and Write) are associated with pages:**
  - If the Read/Write flag of a Page Directory or Page Table entry is equal to 0, the corresponding Page Table or page can only be read;
  - otherwise it can be read and written.
  - Recent Intel Pentium 4 processors support an **NX (No eXecute) flag in each 64-bit Page Table entry**.

2.4.4. An Example of Regular Paging

- A simple example will help in clarifying how regular paging works. Let's assume that the kernel assigns the linear address space between 0x20000000 and 0x2003ffff to a running process.
- This space consists of exactly 64 pages. We don't care about the physical addresses of the page frames containing the pages; in fact, some of them might not even be in main memory. We are interested only in the remaining fields of the Page Table entries.
- Let's start with the 10 most significant bits of the linear addresses assigned to the process, which are interpreted as the Directory field by the paging unit.
  - The addresses start with a 2 followed by zeros, so the 10 bits all have the same value, namely 0x080 or 128 decimal. (binary: 00**00 1000 0000)**
  - Thus the Directory field in all the addresses refers to the 129th entry of the process Page Directory. The corresponding entry must contain the physical address of the Page Table assigned to the process (see Figure 2-9).
  - If no other linear addresses are assigned to the process, all the remaining 1,023 entries of the Page Directory are filled with zeros.
- The values assumed by the intermediate 10 bits, (that is, the values of the Table field) range from 0 to 0x03f, or from 0 to 63 decimal. Thus, only the first 64 entries of the Page Table are valid. The remaining 960 entries are filled with zeros.

An example of paging



An example of paging

- Suppose that the process needs to read the byte at linear address **0x20021406**. This address is handled by the paging unit as follows:
  - The Directory field 0x80 is used to select entry 0x80 of the Page Directory, which points to the Page Table associated with the process's pages.
  - The Table field 0x21 is used to select entry 0x21 of the Page Table, which points to the page frame containing the desired page.
  - Finally, the Offset field 0x406 is used to select the byte at offset 0x406 in the desired page frame.
- If the Present flag of the 0x21 entry of the Page Table is cleared, the **page is not present** in main memory; in this case, the paging unit issues a Page Fault exception while translating the linear address.
- The same exception is issued whenever the process attempts to **access linear addresses outside of the interval** delimited by 0x20000000 and 0x2003ffff, because the Page Table entries not assigned to the process are filled with zeros; in particular, their Present flags are all cleared.

2.4.5. The Physical Address Extension (PAE) Paging Mechanism

- The amount of RAM supported by a processor is limited by the number of address pins connected to the address bus. Older Intel processors from the 80386 to the Pentium used **32-bit physical addresses**. In theory, up to **4 GB of RAM** could be installed on such systems;
  - in practice, due to the **linear address space requirements** of User Mode processes, **the kernel cannot directly address more than 1 GB of RAM**, as we will see in the later section "Paging in Linux."
- However, big servers that need to run hundreds or thousands of processes at the same time require more than 4 GB of RAM.
- Intel has satisfied these requests by **increasing the number of address pins on its processors from 32 to 36**. **Starting with the Pentium Pro, all Intel processors are now able to address up to 2**36 = 64 GB of RAM**.
- With the Pentium Pro processor, Intel introduced a mechanism called Physical Address Extension (PAE).

PAE

- PAE is activated by setting the Physical Address Extension (PAE) flag in the **cr4 control register**.
- Intel has changed the paging mechanism in order to support PAE.
  - The **64 GB of RAM are split into 2**24 distinct page frames**, and the physical address field of Page Table entries has been expanded from 20 to 24 bits.
  - Because a PAE Page Table entry must include the 12 flag bits (described in the earlier section "Regular Paging") and the 24 physical address bits, for a grand total of 36, the **Page Table entry size has been doubled from 32 bits to 64 bits**. As a result, a **4-KB PAE Page Table includes 512 entries instead of 1,024.**
  - A new level of Page Table called the **Page Directory Pointer Table (PDPT)** consisting of **four 64-bit entries** has been introduced.
  - The **cr3 control register** contains a 27-bit **Page Directory Pointer Table base address** field. Because PDPTs are stored in the first 4 GB of RAM and aligned to a multiple of 32 bytes (2**5), 27 bits are sufficient to represent the base address of such tables.
- The Page Size (PS) flag in the page directory entry enables large page sizes (**2 MB when PAE is enabled**).

PAE

- When mapping linear addresses to 4 KB pages (**PS flag cleared** in Page Directory entry), the 32 bits of a linear address are interpreted in the following way:
  - **cr3:** Points to a PDPT
  - **bits 31-30:** Point to 1 of 4 possible entries in PDPT
  - **bits 29-21:** Point to 1 of 512 possible entries in Page Directory
  - **bits 20-12:** Point to 1 of 512 possible entries in Page Table
  - **bits 11-0:** Offset of 4-KB page
- When mapping linear addresses to 2-MB pages (**PS flag set** in Page Directory entry), the 32 bits of a linear address are interpreted in the following way:
  - **cr3:** Points to a PDPT
  - **bits 31-30:** Point to 1 of 4 possible entries in PDPT
  - **bits 29-21:** Point to 1 of 512 possible entries in Page Directory
  - **bits 20-0:** Offset of 2-MB page

To summarize

- Once cr3 is set, it is possible to address up to 4 GB of RAM. If we want to address more RAM, we'll have to put a new value in cr3 or change the content of the PDPT.
- **However, the main problem with PAE is that linear addresses are still 32 bits long. This forces kernel programmers to reuse the same linear addresses to map different areas of RAM.**
- We'll sketch how Linux initializes Page Tables when PAE is enabled in the later section, "Final kernel Page Table when RAM size is more than 4096 MB."
- **Clearly, PAE does not enlarge the linear address space of a process, because it deals only with physical addresses**. Furthermore, only the kernel can modify the page tables of the processes, thus a process running in User Mode cannot use a physical address space larger than 4 GB.
- On the other hand, PAE allows the kernel to exploit up to 64 GB of RAM, and thus to increase significantly the number of processes in the system.

2.4.6. Paging for 64-bit Architectures

- The third level of paging present in 80 x 86 processors with PAE enabled has been introduced only to **lower from 1024 to 512 the number of entries** in the Page Directory and Page Tables.
- This enlarges the Page Table entries **from 32 bits to 64 bits** so that they can store the 24 most significant bits of the physical address.
- Two-level paging, however, is not suitable for computers that adopt a 64-bit architecture. Let's use a thought experiment to explain why:
  - Start by assuming a standard page size of 4 KB. Because 1 KB covers a range of 2^10 addresses, 4 KB covers 2^12 addresses, so the Offset field is 12 bits.
  - This leaves up to 52 bits of the linear address to be distributed between the Table and the Directory fields.
  - If we now decide to use only 48 of the 64 bits for addressing (this restriction leaves us with a comfortable 256 TB address space!), the remaining 48-12 = 36 bits will have to be split among Table and the Directory fields.
  - If we now decide to reserve 18 bits for each of these two fields, both the Page Directory and the Page Tables of each process should include 2^18 entries that is, more than 256,000 entries.

Paging levels in some 64-bit architectures

- All hardware paging systems for 64-bit processors make use of additional paging levels. The **number of levels used depends on the type of processor**. Table 2-4 summarizes the main characteristics of the hardware paging systems used by some 64-bit platforms supported by Linux.

Platform	Page size	address bits used	paging levels	splitting
alpha	8 Kb *	43	3	10 + 10 + 10 + 13
ia64	4 KB *	39	3	9 + 9 + 9 + 12
ppc64	4 KB	41	3	10 + 10 + 9 + 12
sh64	4 KB	41	3	10 + 10 + 9 + 12
x86_64	4 KB	48	4	9 + 9 + 9 + 9 + 12

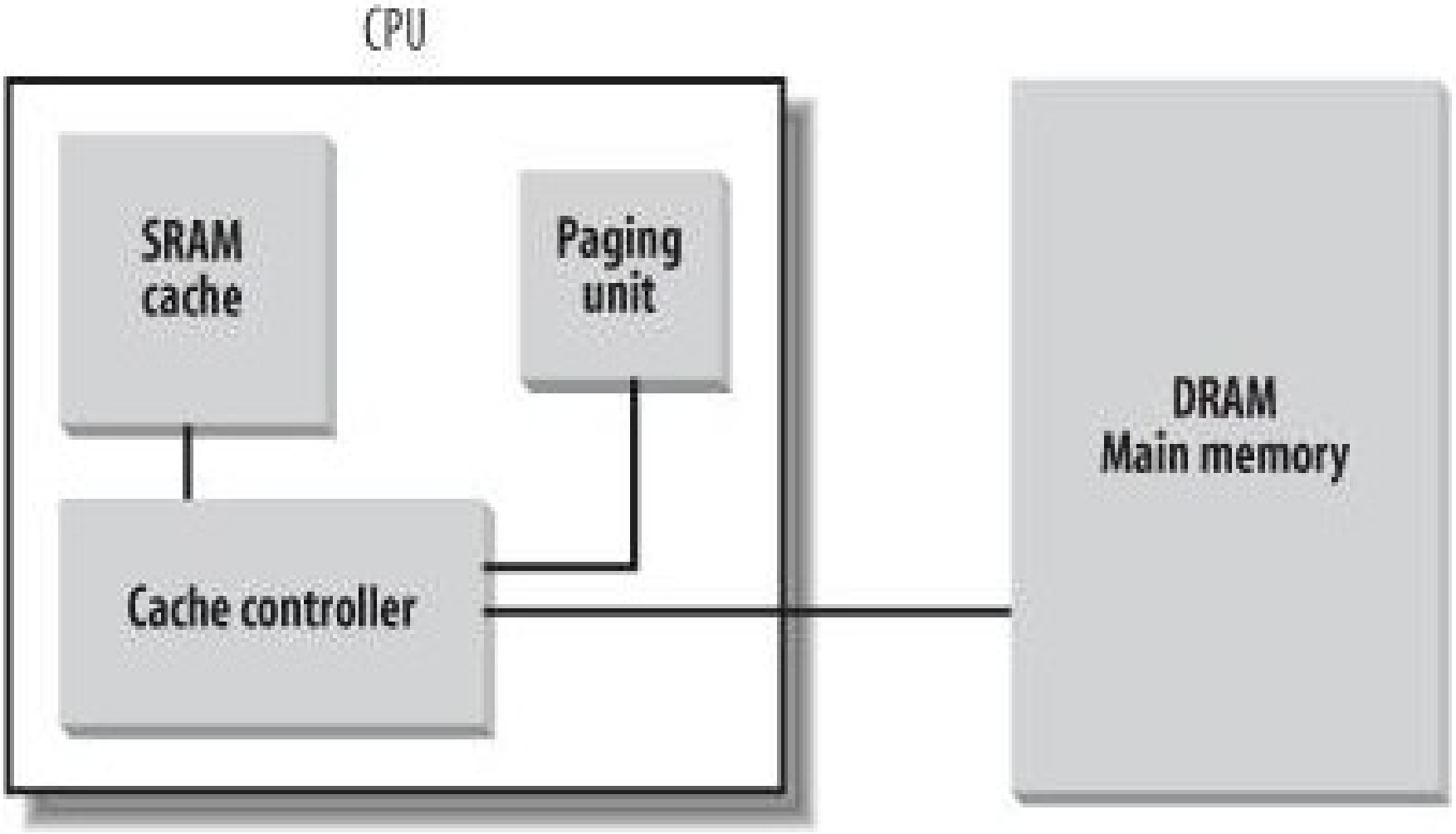
*This architecture supports different page sizes; we select a typical page size adopted by Linux

- As we will see in the section "Paging in Linux" later in this chapter, Linux succeeds in providing a **common paging model** that fits most of the supported hardware paging systems.

2.4.7. Hardware Cache

- Today's microprocessors have clock rates of several gigahertz, while dynamic RAM (DRAM) chips have access times in the range of hundreds of clock cycles.
- This means that the CPU may be held back considerably while executing instructions that require fetching operands from RAM and/or storing results into RAM.
- Hardware cache memories were introduced to reduce the speed mismatch between CPU and RAM.
- They are based on the well-known **locality principle** , which holds both for programs and data structures.
- It therefore makes sense to introduce a **smaller and faster memory** that contains the most recently used code and data.
- For this purpose, a new unit called the **line** was introduced into the 80 x 86 architecture. It consists of a **few dozen contiguous bytes that are transferred in burst mode** between the slow DRAM and the fast on-chip static RAM (SRAM) used to implement caches.

Figure 2-10. Processor hardware cache



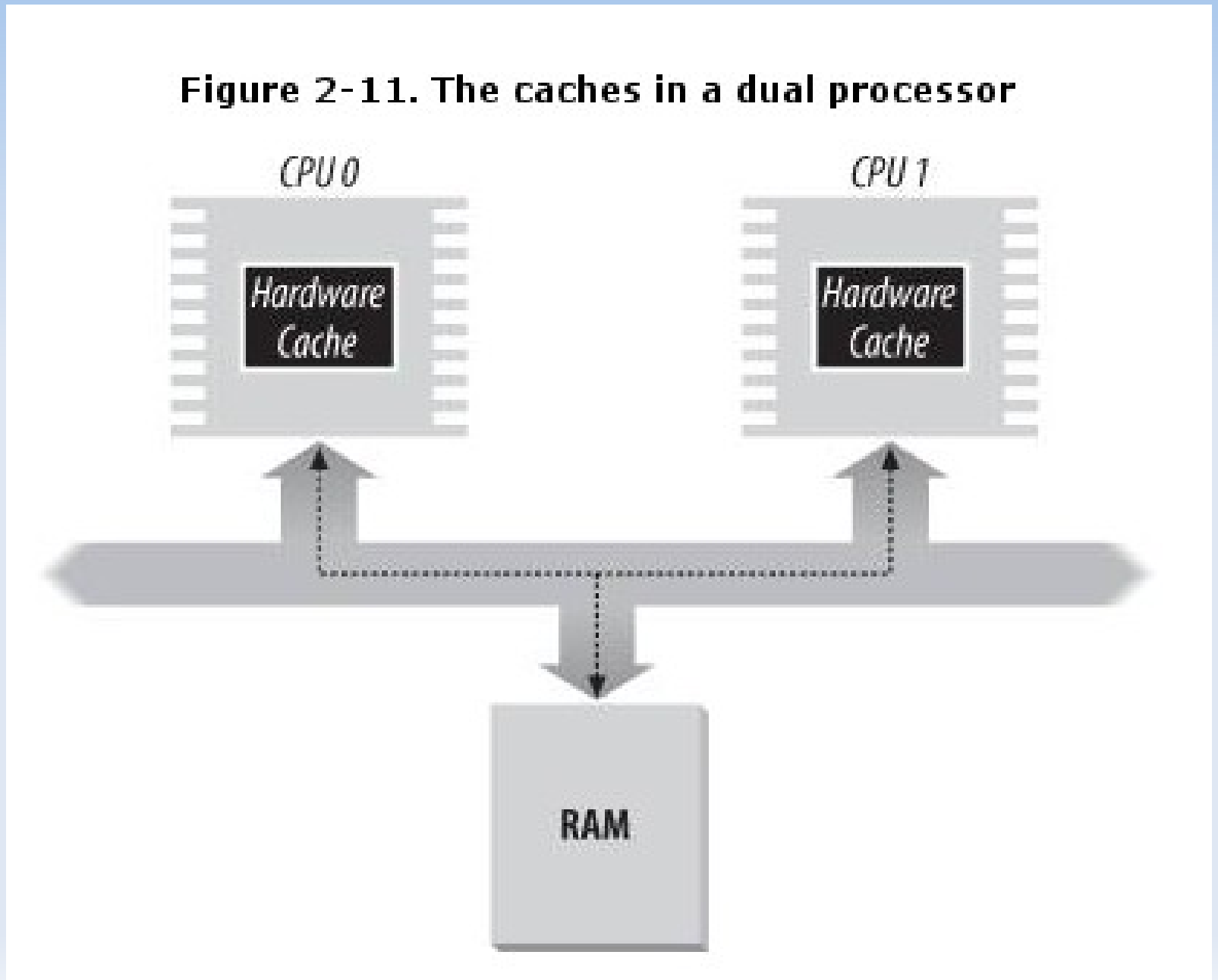
Hit and miss, write-through and write-back

- When accessing a RAM memory cell, the CPU extracts the subset index from the physical address and compares the tags of all lines in the subset with the high-order bits of the physical address. If a line with the same tag as the high-order bits of the address is found, the CPU has a cache hit; otherwise, it has a cache miss.
- When a cache hit occurs, the cache controller behaves differently, depending on the access type.
  - For a **read operation**, the controller selects the data from the cache line and transfers it into a CPU register; the RAM is not accessed and the CPU saves time, which is why the cache system was invented.
  - For a **write operation**, the controller may implement one of two basic strategies called **write-through and write-back** . In a write-through, the controller always writes into both RAM and the cache line, effectively switching off the cache for write operations. In a write-back, which offers more immediate efficiency, only the cache line is updated and the contents of the RAM are left unchanged. After a write-back, of course, the RAM must eventually be updated. The cache controller writes the cache line back into RAM only when the CPU executes an instruction requiring a flush of cache entries or when a FLUSH hardware signal occurs (usually after a cache miss).
- When a cache miss occurs, the cache line is written to memory, if necessary, and the correct line is fetched from RAM into the cache entry.

Cache snooping

- **Multiprocessor systems** have a separate hardware cache for every processor, and therefore they need additional hardware circuitry to synchronize the cache contents.
- As shown in Figure 2-11, each CPU has its own local hardware cache. But now updating becomes more time consuming: whenever a CPU modifies its hardware cache,
- it must check whether the **same data is contained in the other hardware cache**; if so, it must notify the other CPU to update it with the proper value.
- This activity is often called **cache snooping** . Luckily, all this is done at the hardware level and is of no concern to the kernel.

Caches in SMP



Caches and page table entries

- The **CD flag of the cr0** processor register is used to enable or disable the cache circuitry. The **NW flag**, in the same register, specifies whether the write-through or the write-back strategy is used for the caches.

- Another interesting feature of the Pentium cache is that it lets an operating system associate a different cache management policy with each page frame.

  - For this purpose, each Page Directory and each Page Table entry includes two flags: **PCD (Page Cache Disable)**, which specifies whether the cache must be enabled or disabled while accessing data included in the page frame;

  - and **PWT (Page Write-Through)**, which specifies whether the write-back or the write-through strategy must be applied while writing data into the page frame.

- Linux clears the PCD and PWT flags of all Page Directory and Page Table entries; as a result, caching is enabled for all page frames, and the write-back strategy is always adopted for writing.

2.4.8. Translation Lookaside Buffers (TLB)

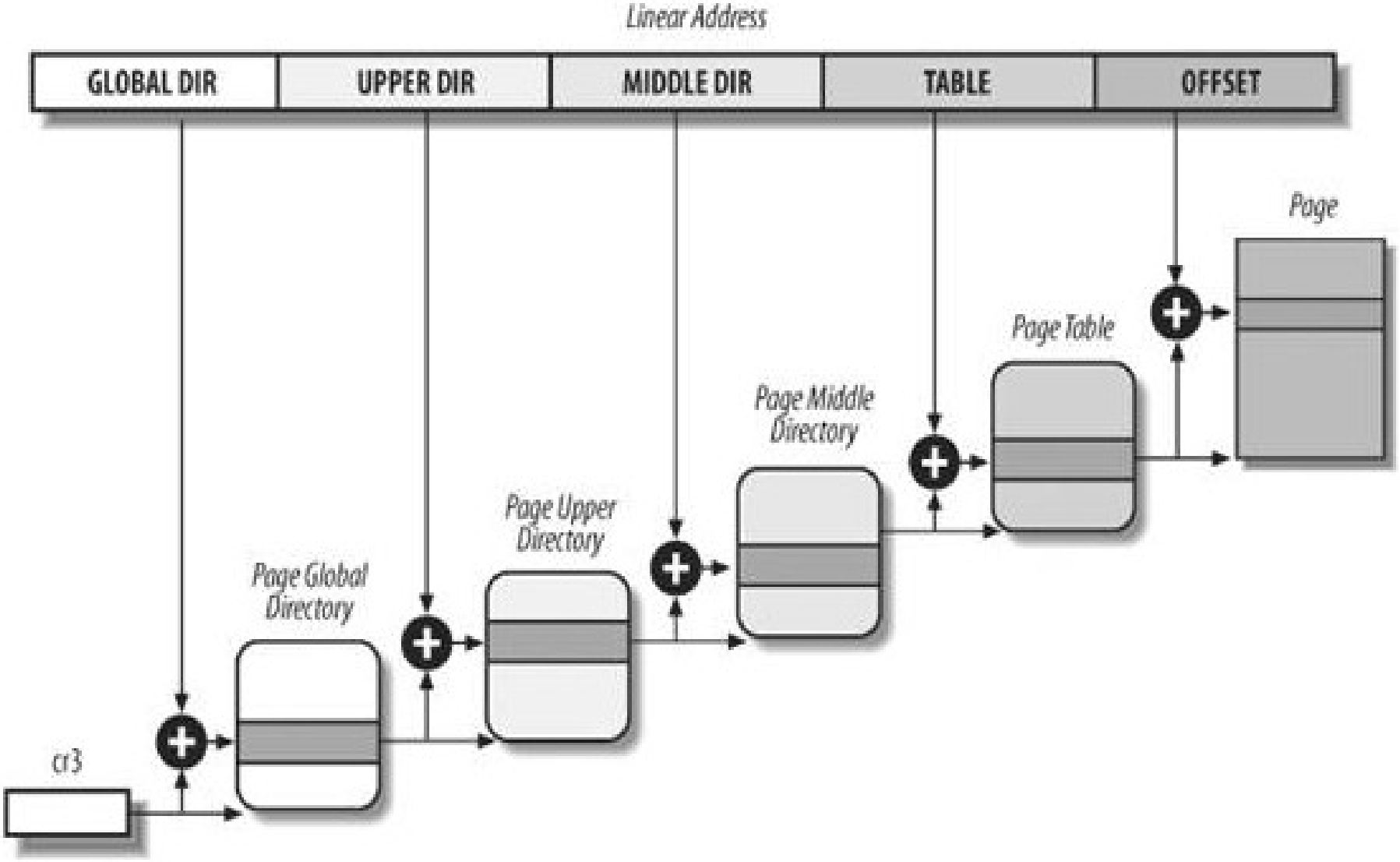
- Besides general-purpose hardware caches, 80 x 86 processors include another cache called Translation Lookaside Buffers (TLB) to speed up linear address translation.
- When a linear address is used for the first time, the corresponding physical address is computed through slow accesses to the Page Tables in RAM. The physical address is then stored in a TLB entry so that further references to the same linear address can be quickly translated.
- In a **multiprocessor system**, each CPU has its own TLB, called the **local TLB** of the CPU. Contrary to the hardware cache, the corresponding entries of the **TLB need not be synchronized**, because processes running on the existing CPUs may associate the same linear address with different physical ones.
- When the **cr3 control register of a CPU is modified**, the hardware automatically **invalidates all entries of the local TLB**, because a new set of page tables is in use and the TLBs are pointing to old data.

Paging in Linux

- Linux adopts a **common paging model** that fits both **32-bit and 64-bit architectures**.
- Up to version 2.6.10, the Linux paging model consisted of three paging levels. **Starting with version 2.6.11, a four-level paging model has been adopted.** This change has been made to fully support the linear address bit splitting used by the x86_64 platform.
- The **four types of page tables** illustrated in Figure 2-12 are called:
  - Page Global Directory
  - Page Upper Directory
  - Page Middle Directory
  - Page Table
- Thus the **linear address can be split into up to five parts**. Figure 2-12 does not show the bit numbers, because **the size of each part depends on the computer architecture**.

The Linux paging model

Figure 2-12. The Linux paging model



paging levels for 32 and 64 bits

- For **32-bit architectures with no Physical Address Extension**, two paging levels are sufficient.
  - Linux essentially eliminates the Page Upper Directory and the Page Middle Directory fields by saying that they contain zero bits. However, the positions of the Page Upper Directory and the Page Middle Directory in the sequence of pointers are kept so that the same code can work on 32-bit and 64-bit architectures.
- For **32-bit architectures with the Physical Address Extension** enabled, three paging levels are used.
  - The Linux's Page Global Directory corresponds to the 80 x 86's Page Directory Pointer Table, the Page Upper Directory is eliminated, the Page Middle Directory corresponds to the 80 x 86's Page Directory, and the Linux's Page Table corresponds to the 80 x 86's Page Table.
- Finally, for **64-bit architectures** three or four levels of paging are used depending on the linear address bit splitting performed by the hardware (see Table 2-2).

paging

- Linux's **handling of processes relies heavily on paging**. In fact, the automatic translation of linear addresses into physical ones makes the following design objectives feasible:
  - Assign a different physical address space to each process, ensuring an **efficient protection against addressing errors**.
  - **Distinguish pages (groups of data) from page frames (physical addresses in main memory)**. This allows the same page to be stored in a page frame, then saved to disk and later reloaded in a different page frame. This is the basic ingredient of the virtual memory mechanism.
- In the remaining part of this chapter, we will refer for the sake of concreteness to the paging circuitry used by the 80 x 86 processors.
- As we will see in Chapter 9, **each process has its own Page Global Directory and its own set of Page Tables**.
- When a **process switch** occurs, Linux **saves the cr3 control register in the descriptor** of the process previously in execution and then loads cr3 with the value stored in the descriptor of the process to be executed next. Thus, when the new process resumes its execution on the CPU, the paging unit refers to the correct set of Page Tables.

2.5.3. Physical Memory Layout

- During the initialization phase the kernel must build a **physical addresses map** that specifies which physical address ranges are usable by the kernel and which are **unavailable** (either because they map hardware devices' I/O shared memory or because the corresponding page frames contain BIOS data).
- The kernel considers the following page frames as reserved :
  - Those falling in the unavailable physical address ranges
  - Those containing the kernel's code and initialized data structures
- A page contained in a reserved page frame can never be dynamically assigned or swapped to disk.

Kernel installed in 0x00100000

- As a general rule, the Linux kernel is installed in RAM starting from the physical address 0x00100000 i.e., from the second megabyte.
- The total number of page frames required depends on how the kernel is configured. A typical configuration yields a kernel that can be loaded in less than 3 MB of RAM.
- Why isn't the kernel loaded starting with the first available megabyte of RAM? Well, the PC architecture has several peculiarities that must be taken into account. For example:
  - Page frame 0 is used by BIOS to store the system hardware configuration detected during the Power-On Self-Test(POST); the BIOS of many laptops, moreover, writes data on this page frame even after the system is initialized.
  - Physical addresses ranging from 0x000a0000 to 0x000fffff are usually reserved to BIOS routines and to map the internal memory of ISA graphics cards. This area is the well-known hole from 640 KB to 1 MB in all IBM-compatible PCs: the physical addresses exist but they are reserved, and the corresponding page frames cannot be used by the operating system.
  - Additional page frames within the first megabyte may be reserved by specific computer models. For example, the IBM ThinkPad maps the 0xa0 page frame into the 0x9f one.

Physical addresses map

- In the early stage of the boot sequence (see Appendix A), the kernel queries the **BIOS** and learns the **size of the physical memory**.
- In recent computers, the kernel also invokes a **BIOS** procedure to build a **list of physical address ranges** and their corresponding memory types. (http://wiki.osdev.org/How_Do_I_Determine_The_Amount_Of_RAM)
- Later, the kernel executes the **machine_specific_memory_setup( )** function, which builds the physical addresses map (see Table 2-9 for an example).
- Of course, the kernel builds this table on the basis of the BIOS list, if this is available; otherwise the kernel builds the table following the conservative default setup: all page frames with numbers from 0x9f (LOWMEMSIZE( )) to 0x100 (HIGH_MEMORY) are marked as reserved.

Table 2-9. Example of BIOS-provided physical addresses map

Start	End	Type
0x00000000	0x0009ffff	Usable
0x000f0000	0x000fffff	Reserved
0x00100000	0x07feffff	Usable
0x07ff0000	0x07ff2fff	ACPI data
0x07ff3000	0x07ffffff	ACPI NVS
0xffff0000	0xffffffff	Reserved

A typical configuration for a computer having 128 MB of RAM is shown in Table 2-9.

- The physical address range from 0x07ff0000 to 0x07ff2fff stores information about the hardware devices of the system written by the BIOS in the POST phase; during the initialization phase, the kernel copies such information in a suitable kernel data structure, and then considers these page frames usable.
- Conversely, the physical address range of 0x07ff3000 to 0x07ffffff is mapped to ROM chips of the hardware devices.
- The physical address range starting from 0xffff0000 is marked as reserved, because it is mapped by the hardware to the BIOS's ROM chip (see Appendix A).
- Notice that the **BIOS may not provide information for some physical address ranges** (in the table, the range is 0x000a0000 to 0x000effff). To be on the safe side, Linux assumes that such ranges are not usable.

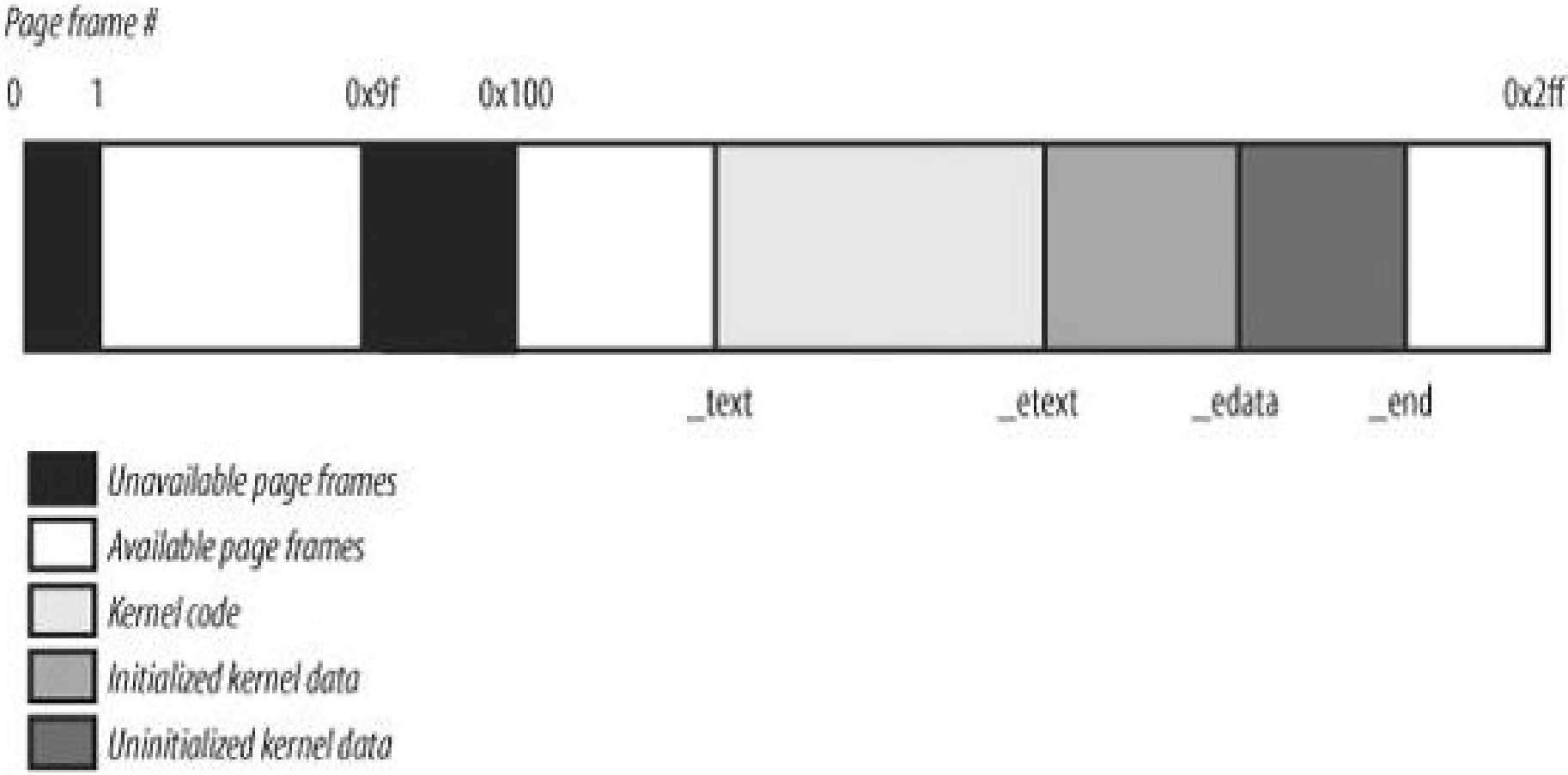
Table 2-10. Variables describing the kernel's physical memory layout

The **setup_memory( )** function is invoked right after **machine_specific_memory_setup( )**: it analyzes the table of physical memory regions and initializes a few variables that describe the kernel's physical memory layout.

Variable name	Description
num_physpages	Page frame number of the highest usable page frame total
ram_pages	Total number of usable page frames
min_low_pfn	Page frame number of the first usable page frame after the kernel mage in RAM
max_pfn	Page frame number of the last usable page frame
max_low_pfn	Page frame number of the last page frame directly mapped by the kernel (low mem)
totalhigh_pages	Total number of page frames not directly mapped by the kernel (high mem)
highstart_pfn	Page frame number of the first page frame not directly mapped by the kernel
highend_pfn	Page frame number of the last page frame not directly mapped by the kernel

Skip first megabyte

Figure 2-13. The first 768 page frames (3 MB) in Linux 2.6



2.5.4. Process Page Tables

- The linear address space of a process is divided into two parts:
  - Linear addresses from 0x00000000 to 0xbfffffff can be addressed when the process runs in either User or Kernel Mode.
  - Linear addresses from 0xc0000000 to 0xffffffff can be addressed only when the process runs in Kernel Mode.
- When a process runs in User Mode, it issues linear addresses smaller than 0xc0000000;
- when it runs in Kernel Mode, it is executing kernel code and the linear addresses issued are greater than or equal to 0xc0000000. In some cases, however, the kernel must access the User Mode linear address space to retrieve or store data.

0XC000000

- The PAGE_OFFSET macro yields the value 0xc0000000; this is the offset in the linear address space of a process where the kernel lives.

- The content of the first entries of the Page Global Directory that map linear addresses lower than 0xc0000000 (the first 768 entries with PAE disabled, or the first 3 entries with PAE enabled) depends on the specific process.

- Conversely, the remaining entries should be the same for all processes and equal to the corresponding entries of the **master kernel Page Global Directory** (see the following section).

2.5.5. Kernel Page Tables

- The kernel maintains a set of page tables for its own use, rooted at a so-called **master kernel Page Global Directory**. After system initialization, this set of page tables is never directly used by any process or kernel thread; rather, the highest entries of the master kernel Page Global Directory are the reference model for the corresponding entries of the Page Global Directories of every regular process in the system.

- We now describe how the kernel initializes its own page tables. This is a two-phase activity. In fact, right after the kernel image is loaded into memory, the CPU is still running in real mode; thus, paging is not enabled.

- In the first phase, the kernel creates a limited address space including the kernel's code and data segments, the initial Page Tables, and 128 KB for some dynamic data structures. This minimal address space is just large enough to install the kernel in RAM and to initialize its core data structures.

- In the second phase, the kernel takes advantage of all of the existing RAM and sets up the page tables properly. Let us examine how this plan is executed.

2.5.5.1. Provisional kernel Page Tables

- A provisional Page Global Directory is initialized statically during kernel compilation, while the provisional Page Tables are initialized by the startup_32( ) assembly language function defined in arch/i386/kernel/head_32.S

- We won't bother mentioning the Page Upper Directories and Page Middle Directories anymore, because they are equated to Page Global Directory entries. PAE support is not enabled at this stage.

- The provisional Page Global Directory is contained in the swapper_pg_dir variable. The provisional Page Tables are stored starting from pg0, right after the end of the kernel's uninitialized data segments (symbol _end in Figure 2-13).

- For the sake of simplicity, **let's assume that the kernel's segments, the provisional Page Tables, and the 128 KB memory area fit in the first 8 MB of RAM**. In order to map 8 MB of RAM, two Page Tables are required.

- The objective of this first phase of paging is to allow these 8 MB of RAM to be easily addressed both in real mode and protected mode.

- Identity mapping + linear mapping

Provisional kernel Page Tables

- The kernel must create a mapping from both the linear addresses 0x00000000 through 0x007fffff and the linear addresses 0xc0000000 through 0xc07fffff into the physical addresses 0x00000000 through 0x007fffff.
- In other words, the kernel during its first phase of initialization can address the first 8 MB of RAM by either linear addresses identical to the physical ones or 8 MB worth of linear addresses, starting from 0xc0000000.
- The Kernel creates the desired mapping by filling all the swapper_pg_dir entries with zeroes, except for entries 0, 1, 0x300 (decimal 768), and 0x301 (decimal 769); the latter two entries span all linear addresses between 0xc0000000 and 0xc07fffff. The 0, 1, 0x300, and 0x301 entries are initialized as follows:
  - The address field of entries 0 and 0x300 is set to the physical address of pg0, while the address field of entries 1 and 0x301 is set to the physical address of the page frame following pg0.
  - The Present, Read/Write, and User/Supervisor flags are set in all four entries.
  - The Accessed, Dirty, PCD, PWD, and Page Size flags are cleared in all four entries.

Enabling paging

- The startup_32( ) assembly language function also enables the paging unit.
- This is achieved by **loading the physical address of swapper_pg_dir into the cr3** control register
- and by setting the **PG flag of the cr0** control register, as shown in the following equivalent code fragment:

```
movl $swapper_pg_dir - 0xc0000000,%eax
movl %eax,%cr3       /* set the page table pointer.. */
movl %cr0,%eax
orl $0x80000000,%eax
movl %eax,%cr0       /* ..and set paging (PG) bit */
```

2.5.7. Handling the Hardware Cache and the TLB

- The last topic of memory addressing deals with how the kernel makes an optimal use of the hardware caches.
- Hardware caches and Translation Lookaside Buffers play a crucial role in boosting the performance of modern computer architectures.
- Several techniques are used by kernel developers to reduce the number of cache and TLB misses.

2.5.7.1. Handling the hardware cache

- Hardware caches are addressed by cache lines. The L1_CACHE_BYTES macro yields the size of a cache line in bytes.
  - On Intel models earlier than the Pentium 4, the macro yields the value 32; on a Pentium 4, it yields the value 128.
- To **optimize the cache hit rate**, the **kernel considers the architecture** in making the following decisions.
  - The most frequently used **fields of a data structure** are placed at the low offset within the data structure, so they can be **cached in the same line**.
  - When allocating a large set of data structures, the kernel tries to store each of them in memory in such a way that all **cache lines are used uniformly**.
- **Cache synchronization is performed automatically** by the 80 x 86 microprocessors, thus the Linux kernel for this kind of processor does not perform any hardware cache flushing.
- **The kernel does provide, however, cache flushing interfaces for processors that do not synchronize caches.**

2.5.7.2. Handling the TLB

- **Processors cannot synchronize their own TLB cache automatically** because it is the kernel, and not the hardware, that decides when a mapping between a linear and a physical address is no longer valid.
- **Intel microprocessors offers only two TLB-invalidating techniques:**
  - All Pentium models automatically flush the TLB entries relative to non-global pages when a **value is loaded into the cr3** register.
  - In Pentium Pro and later models, the **invlpg assembly language instruction** invalidates a single TLB entry mapping a given linear address.
- Table 2-12 lists the Linux macros that exploit such hardware techniques;
- these macros are the basic ingredients to implement architecture-independent methods

Table 2-12. TLB-invalidating macros for the Intel Pentium Pro and later processors

Macro name	Description
_ _flush_tlb( )	Rewrites cr3 register back into itself
_ _flush_tlb_global( )	Disables global pages by clearing the PGE flag of cr4, rewrites cr3 register back into itself, and sets again the PGE flag
_ _flush_tlb_single(addr)	Executes invlpg assembly language instruction with parameter addr

SMPs and avoiding TLB flushes

- The architecture-independent TLB-invalidating methods are extended quite simply to multiprocessor systems. The function running on a CPU sends an **Interprocessor Interrupt** to the other CPUs that forces them to execute the proper TLB-invalidating function.
- As a **general rule**, any process switch implies changing the set of active page tables.
- The kernel succeeds, however, in **avoiding TLB flushes** in the following cases:
  - When performing a process switch between two **regular processes** that use the **same set of page tables**.
  - When performing a process switch between a regular process and a **kernel thread**.

More cases of flushes and lazy TLB mode

- Besides process switches, there are **other cases** in which the kernel needs to flush some entries in a TLB.
  - For instance, **when the kernel assigns a page frame to a User Mode** process and stores its physical address into a Page Table entry, **it must flush any local TLB entry** that refers to the corresponding linear address.
  - On **multiprocessor systems**, the kernel also must flush the **same TLB entry on the CPUs** that are using the same set of page tables, if any.
- To avoid useless TLB flushing in multiprocessor systems, the kernel uses a technique called **lazy TLB mode** .
  - The basic idea is the following: if several CPUs are using the same page tables and a TLB entry must be flushed on all of them, then TLB flushing may, in some cases, be delayed on CPUs running kernel threads.