Memory Addressing

Memory Management

- This chapter deals with addressing techniques.
- An operating system is not forced to keep track of physical memory all by itself.

Brief Overview of Memory Managment

- Memory management is by far the most complex activity in a Unix kernel.
- All recent Unix systems provide a useful abstraction called virtual memory.
- Virtual memory acts as a logical layer between the application memory requests and the hardware Memory Management Unit (MMU).
- Virtual memory has many purposes and advantages:
- Several processes can be executed concurrently.
- It is possible to run applications whose memory needs are larger than the available physical memory.
- Processes can execute a program whose code is only partially loaded in memory.
- Each process is allowed to access a subset of the available physical memory.
- Processes can share a single memory image of a library or program.
- Programs can be relocatable, that is, they can be placed anywhere in physical memory.
- Programmers can write machine-independent code, since they do not need to be concerned about physical memory organization.

- The main ingredient of a virtual memory subsystem is the notion of virtual address space.
- The set of memory references that a process can use is different from physical memory addresses.
- When a process uses a virtual address, the kernel and the MMU cooperate to locate the actual physical location of the requested memory item.
- Today's CPUs include hardware circuits that automatically translate the virtual addresses into physical ones.
- The available RAM is partitioned into page frames 4 or 8 KB in length, and a set of page tables is introduced to specify the correspondence between virtual and physical addresses.
- These circuits make memory allocation simpler, since a request for a block of contiguous virtual addresses can be satisfied by allocating a group of page frames having noncontiguous physical addresses.

Random Access Memory

- All Unix operating systems clearly distinguish two portions of the random access memory (RAM).
- A few megabytes are dedicated to storing the kernel image (i.e., the kernel code and the kernel static data structures).
- The remaining portion of RAM is usually handled by the virtual memory system and is used in three possible ways:

  - To satisfy kernel requests for buffers, descriptors, and other dynamic kernel data structures.
  - To satisfy process requests for generic memory areas and for memory mapping of files
  - To get better performance from disks and other buffered devices by means of caches

- Since the available RAM is limited, some balancing among request types must be done, particularly when little available memory is left.
- When some critical threshold of available memory is reached and a page-frame reclaiming algorithm is invoked to free additional memory.
- One major problem that must be solved by the virtual memory system is memory fragmentation .
- Ideally, a memory request should fail only when the number of free page frames is too small.
- However, the kernel is often forced to use physically contiguous memory areas, hence the memory request could fail even if there is enough memory available but it is not available as one contiguous chunk.

Kernel Memory Allocator

- The Kernel Memory Allocator (KMA) is a subsystem that tries to satisfy the requests for memory areas from all parts of the system.
- A good KMA should have the following features:
  - It must be fast. Actually, this is the most crucial attribute, since it is invoked by all kernel subsystems (including the interrupt handlers).
  - It should minimize the amount of wasted memory.
  - It should try to reduce the memory fragmentation problem.
  - It should be able to cooperate with the other memory management subsystems in order to borrow and release page frames from them.
- Several kinds of KMAs have been proposed, based on a variety of different algorithmic techniques, including:
  - Resource map allocator
  - Power-of-two free lists
  - McKusick-Karels allocator
  - Buddy system
  - Mach's Zone allocator
  - Dynix allocator
  - Solaris's Slab allocator

Process Virtual Address space handling

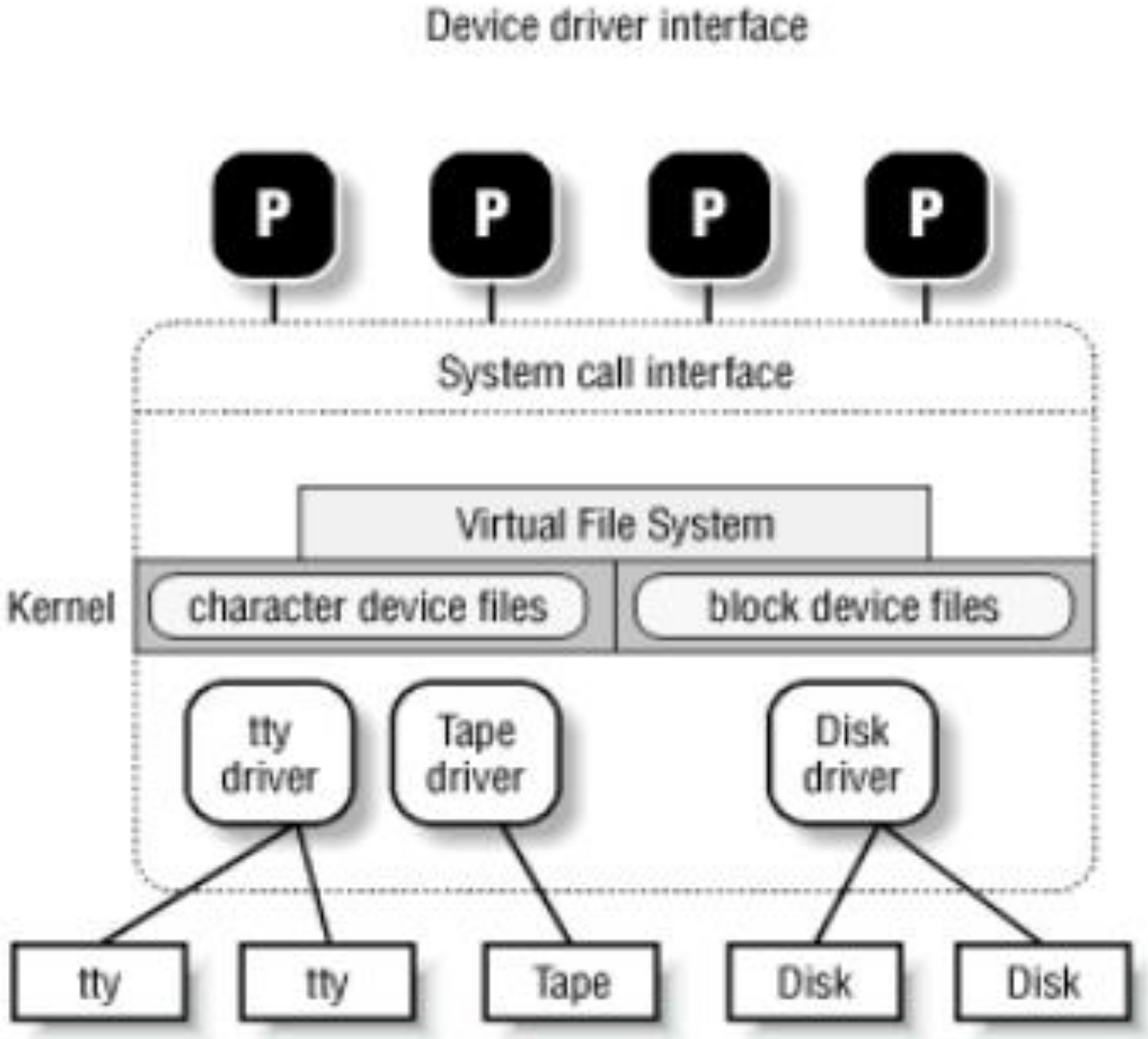
- The address space of a process contains all the virtual memory addresses that the process is allowed to reference.
- The kernel usually stores a process virtual address space as a list of memory area descriptors.
- When a process starts the execution of some program via an exec( )-like system call, the kernel assigns to the process a virtual address space that comprises memory areas for:
  - The executable code of the program
  - The initialized data of the program
  - The uninitialized data of the program
  - The initial program stack (that is, the User Mode stack)
  - The executable code and data of needed shared libraries
  - The heap (the memory dynamically requested by the program)
- All recent Unix operating systems adopt a memory allocation strategy called demand paging.
- With demand paging, a process can start program execution with none of its pages in physical memory.
- As it accesses a nonpresent page, the MMU generates an exception; the exception handler finds the affected memory region, allocates a free page, and initializes it with the appropriate data.

Swapping and Caching

- In order to extend the size of the virtual address space usable by the processes, the Unix operating system makes use of swap areas on disk.
- The virtual memory system regards the contents of a page frame as the basic unit for swapping.
- Whenever some process refers to a swapped-out page, the MMU raises an exception.
- The exception handler then allocates a new page frame and initializes the page frame with its old contents saved on disk.
- On the other hand, physical memory is also used as cache for hard disks and other block devices.
- This is because hard drives are very slow: a disk access requires several milliseconds, which is a very long time compared with the RAM access time.
- Therefore, disks are often the bottleneck in system performance.

Device Drivers

- The kernel interacts with I/O devices by means of device drivers. Device drivers are included in the kernel and consist of data structures and functions that control one or more devices, such as hard disks, keyboards, mouses, monitors, network interfaces, and devices connected to a SCSI bus.
- Each driver interacts with the remaining part of the kernel (even with other drivers) through a specific interface.
- This approach has the following advantages:
  - Device-specific code can be encapsulated in a specific module.
  - Vendors can add new devices without knowing the kernel source code: only the interface specifications must be known.
  - The kernel deals with all devices in a uniform way and accesses them through the same interface.
  - It is possible to write a device driver as a module that can be dynamically loaded in the kernel without requiring the system to be rebooted. It is also possible to dynamically unload a module that is no longer needed, thus minimizing the size of the kernel image stored in RAM.


Device driver interface
P
P
P
P
System call interface
Virtual File System
Kernel
character device files
block device files
tty driver
Tape driver
Disk driver
tty
tty
Tape
Disk
Disk

Memory Addresses

- Programmers casually refer to a memory address as the way to access the contents of a memory cell.
- We have to distinguish among three kinds of addresses:
  - Logical address - Included in the machine language instructions to specify the address of an operand or of an instruction. Each logical address consists of a segment and an offset that denotes the distance from the start of the segment to the actual address.
  - Linear address - A single 32-bit unsigned integer that can be used to address up to 4 GB, that is, up to 4,294,967,296 memory cells.
  - Physical addresses - Used to address memory cells included 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 unsigned integers.

Segmentation in Hardware

- Intel microprocessors perform address translation in two different ways called real mode and protected mode.
- Real mode exists mostly to maintain processor compatibility with older models and to allow the operating system to bootstrap.
- Focus is on protected mode.

Segmentation Registers

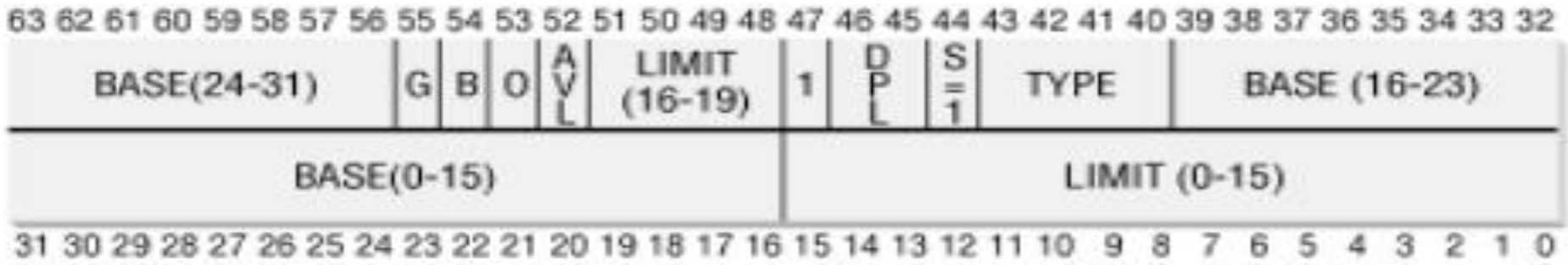
- A logical address consists of two parts: a segment identifier(16 bit - Segment Selector) and an offset (32 bit) that specifies the relative address within the segment.
- To make it easy to retrieve segment selectors quickly, the processor provides segmentation registers whose only purpose is to hold Segment Selectors; these registers are called cs, ss, ds, es, fs, and gs.
- A program can reuse the same segmentation register for different purposes by saving its content in memory and then restoring it later.
- 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 static and external data
- The remaining three segmentation registers are general purpose and may refer to arbitrary segments.

- 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 denotes the highest privilege level, while the value 3 denotes the lowest one. Linux uses only levels and 3, which are respectively called Kernel Mode and User Mode.

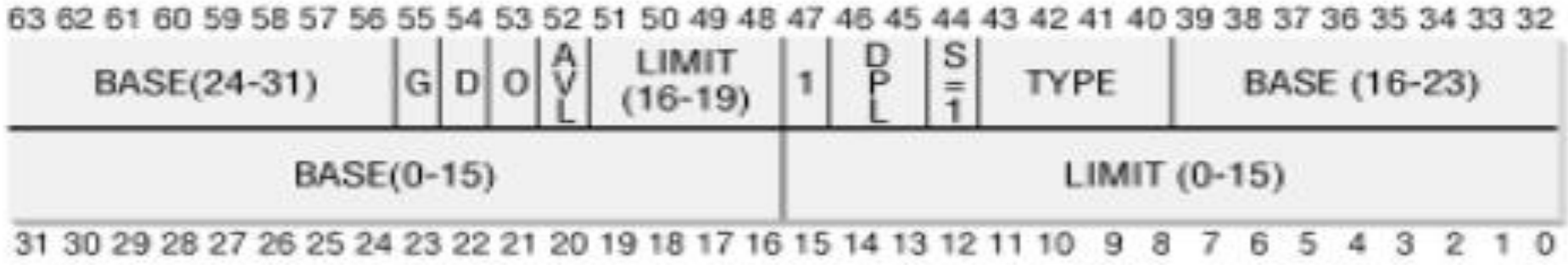
Segment Descriptors

- 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 may have its own LDT. The address of the GDT in main memory is contained in the gdtr processor register and the address of the currently used LDT is contained in the ldtr processor register.

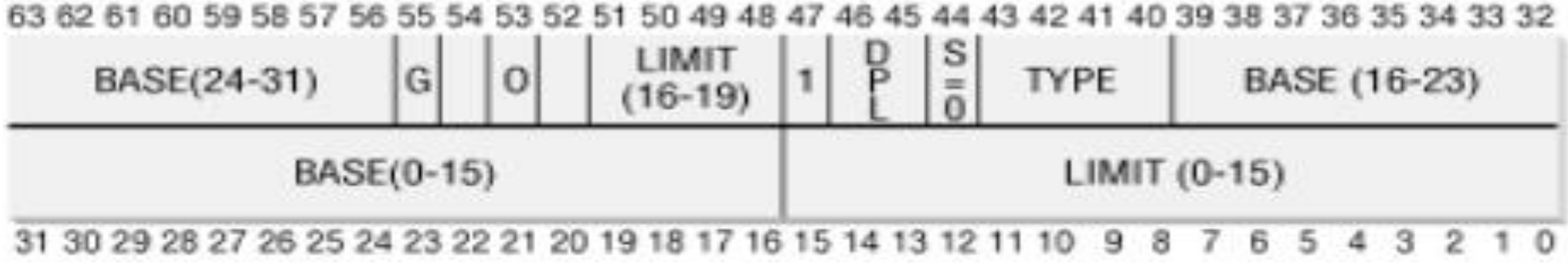
Data Segment Descriptor



Code Segment Descriptor



System Segment Descriptor



- Each Segment Descriptor consists of the following fields:
- A 32-bit Base field that contains the linear address of the first byte of the segment.
- A G granularity flag: if it is cleared, the segment size is expressed in bytes; otherwise, it is expressed in multiples of 4096 bytes.
- A 20-bit Limit field that denotes the segment length in bytes. If G is set to 0, the size of a non-null segment may vary between 1 byte and 1 MB; otherwise, it may vary between 4 KB and 4 GB.
- An S system flag: if it is cleared, the segment is a system segment that stores kernel data structures; otherwise, it is a normal code or data segment.
- A 4-bit Type field that characterizes the segment type and its access rights.

  The following Segment Descriptor types are widely used:

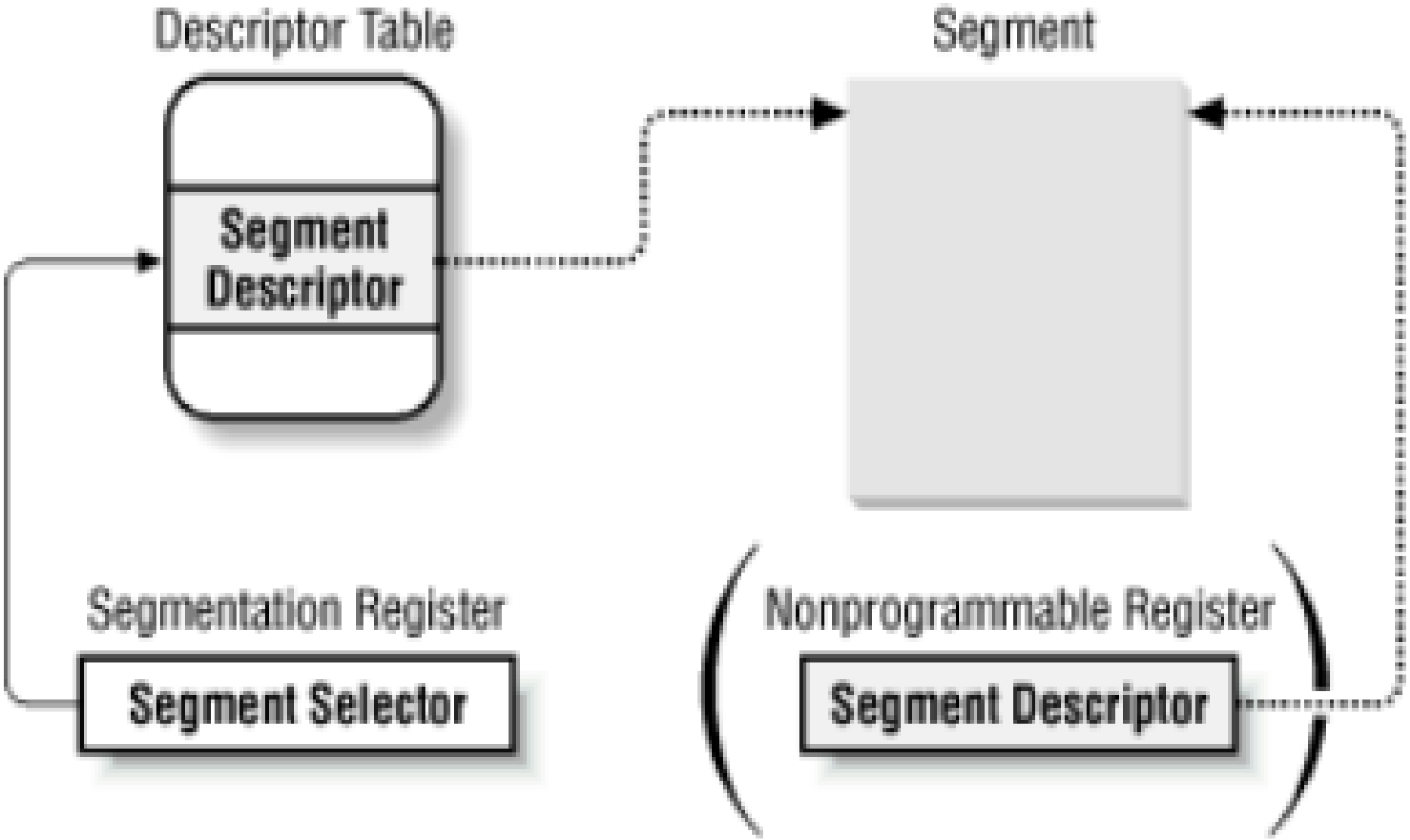
  - Code segment descriptor
  - Data segment descriptor
  - Task State Segment descriptor
  - Local descriptor Table descriptor

- 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.
- A DPL (Descriptor Privilege Level) 2-bit field 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 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.
- A Segment-Present flag that is set to if the segment is currently not stored in main memory.
- Linux always sets this field to 1, since it never swaps out whole segments to disk.
- An additional flag 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 if the addresses used as segment offsets are 32 bits long and it is cleared if they are 16 bits long .
- A reserved bit (bit 53) always set to 0.
- An AVL flag that may be used by the operating system but is ignored in Linux.

Segment Selectors

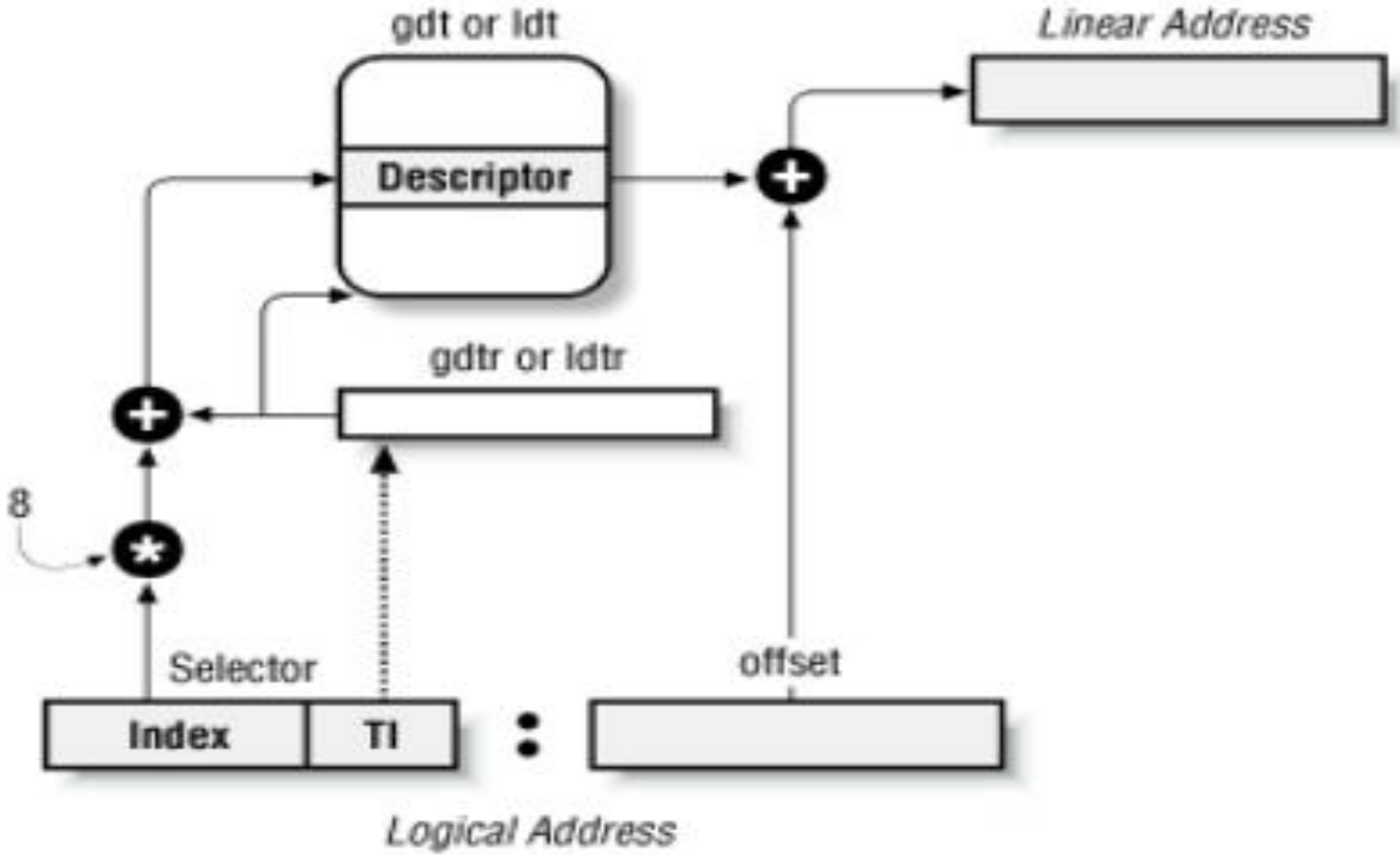
- To speed up the translation of logical addresses into linear addresses, the Intel processor provides an additional nonprogrammable register.
- Each nonprogrammable register contains the 8-byte Segment Descriptor specified by the Segment Selector contained in the corresponding segmentation register.
- Every time a Segment Selector is loaded in a segmentation register, the corresponding Segment Descriptor is loaded from memory into the matching nonprogrammable CPU register.
- From then on, translations of logical addresses referring to that segment can be performed without accessing the GDT or LDT stored in main memory; the processor can just refer directly to the CPU register containing the Segment Descriptor.
- Accesses to the GDT or LDT are necessary only when the contents of the segmentation register change.

- Each Segment Selector includes the following fields:
  - A 13-bit index that identifies the corresponding Segment Descriptor entry contained in the GDT or in the LDT.
  - A TI (Table Indicator) flag that specifies whether the Segment Descriptor is included in the GDT (TI = 0) or in the LDT (TI = 1).
  - An RPL (Requestor Privilege Level) 2-bit field, which is precisely the Current Privilege Level of the CPU when the corresponding Segment Selector is loaded into the cs register.
- Since a Segment Descriptor is 8 bytes long, its relative address inside the GDT or the LDT is obtained by multiplying the most significant 13 bits of the Segment Selector by 8.
- The first entry of the GDT is always set to 0: this ensures that logical addresses with a null Segment Selector will be considered invalid, thus causing a processor exception.
- The maximum number of Segment Descriptors that can be stored in the GDT is thus 8191, that is, 2^{13}-1.


Descriptor Table
Segment
Segment Descriptor
Segmentation Register
Segment Selector
Nonprogrammable Register
Segment Descriptor

Segmentation Unit

- Here, 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, in order 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 to the Base field of the Segment Descriptor the offset of the logical address, thus obtains the linear address.


gdt or ldt
Linear Address
Descriptor
gdtr or ldtr
8
Selector
offset
Index
TI
Logical Address

Segmentation in Linux

- Segmentation has been included in Intel microprocessors to encourage programmers to split their applications in logically related entities, such as subroutines or global and local data areas.
- Linux uses segmentation in a very limited way.
- Segmentation and paging are somewhat redundant since 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 the most popular architectures; however, several RISC processors support segmentation in a very limited way.

- All processes use the same logical addresses, so the total number of segments to be defined is quite limited and it is possible to store all Segment Descriptors in the Global Descriptor Table (GDT).
- This table is implemented by the array gdt_table referred by the gdt variable.
- Local Descriptor Tables are not used by the kernel, although a system call exists that allows processes to create their own LDTs.
- Here are the segments used by Linux:
- A kernel code segment. The fields of the corresponding Segment Descriptor in the GDT have the following values:
    - Base = 0x00000000
    - Limit = 0xfffff
    - G (granularity flag) = 1, for segment size expressed in pages
    - S (system flag) = 1, for normal code or data segment
    - Type = 0xa, for code segment that can be read and executed
    - DPL (Descriptor Privilege Level) = 0, for Kernel Mode
    - D/B (32-bit address flag) = 1, for 32-bit offset addresses

- Thus, the linear addresses associated with that segment start at and reach the addressing limit of 232 - 1.
- The S and Type fields specify that the segment is a code segment that can be read and executed.
- Its DPL value is 0, thus it can be accessed only n Kernel Mode.
- The corresponding Segment Selector is defined by the __KERNEL_CS macro: in order to address the segment, the kernel just loads the value yielded by the macro into the cs register.
- A kernel data segment. The fields of the corresponding Segment Descriptor in the GDT have the following values:
  - Base = 0x00000000
  - Limit = 0xfffff
  - G (granularity flag) = 1, for segment size expressed in pages
  - S (system flag) = 1, for normal code or data segment
  - Type = 2, for data segment that can be read and written
  - DPL (Descriptor Privilege Level) = 0, for Kernel Mode
  - D/B (32-bit address flag) = 1, for 32-bit offset addresses

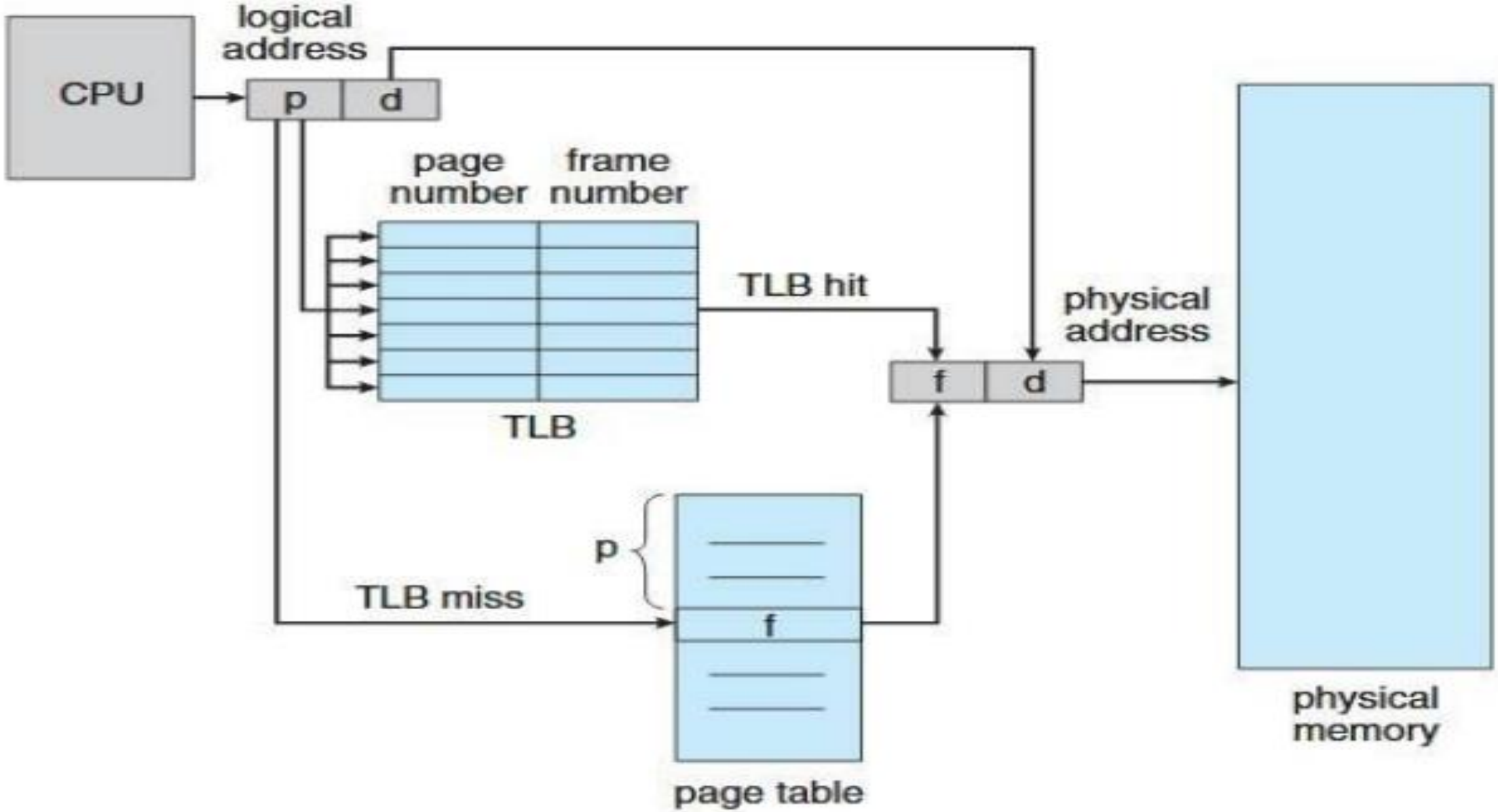
- A user code segment shared by all processes in User Mode. The fields of the corresponding Segment Descriptor in the GDT have the following values:
  - Base = 0x00000000
  - Limit = 0xfffff
  - G (granularity flag) = 1, for segment size expressed in pages
  - S (system flag) = 1, for normal code or data segment
  - Type = 0xa, for code segment that can be read and executed
  - DPL (Descriptor Privilege Level) = 3, for User Mode
  - D/B (32-bit address flag) = 1, for 32-bit offset addresses
- The S and DPL fields specify that the segment is not a system segment and that its privilege level is equal to 3; it can thus be accessed both in Kernel Mode and in User Mode. The corresponding Segment Selector is defined by the __USER_CS macro.

- A user data segment shared by all processes in User Mode. The fields of the corresponding Segment Descriptor in the GDT have the following values:
  - Base = 0x00000000
  - Limit = 0xfffff
  - G (granularity flag) = 1, for segment size expressed in pages
  - S (system flag) = 1, for normal code or data segment
  - Type = 2, for data segment that can be read and written
  - DPL (Descriptor Privilege Level) = 3, for User Mode
  - D/B (32-bit address flag) = 1, for 32-bit offset addresses
- This segment overlaps the previous one: they are identical, except for the value of Type. The corresponding Segment Selector is defined by the __USER_DS macro.

- A Task State Segment (TSS) segment for each process. The descriptors of these segments are stored in the GDT. The Base field of the TSS descriptor associated with each process contains the address of the tss field of the corresponding process descriptor.
- The G flag is cleared, while the Limit field is set to 0xeb, since the TSS segment is 236 bytes long. The Type field is set to 9 or 11 (available 32-bit TSS), and the DPL is set to 0, since processes in User Mode are not allowed to access TSS segments.
- A default LDT segment that is usually shared by all processes. This segment is stored in the default_ldt variable. The default LDT includes a single entry consisting of a null Segment Descriptor.
- Each process has its own LDT Segment Descriptor, which usually points to the common default LDT segment. The Base field is set to the address of default_ldt and the Limit field is set to 7. If a process requires a real LDT, a new 4096-byte segment is created (it can include up to 511 Segment Descriptors), and the default LDT Segment Descriptor associated with that process is replaced in the GDT with a new descriptor with specific values for the Base and Limit fields.

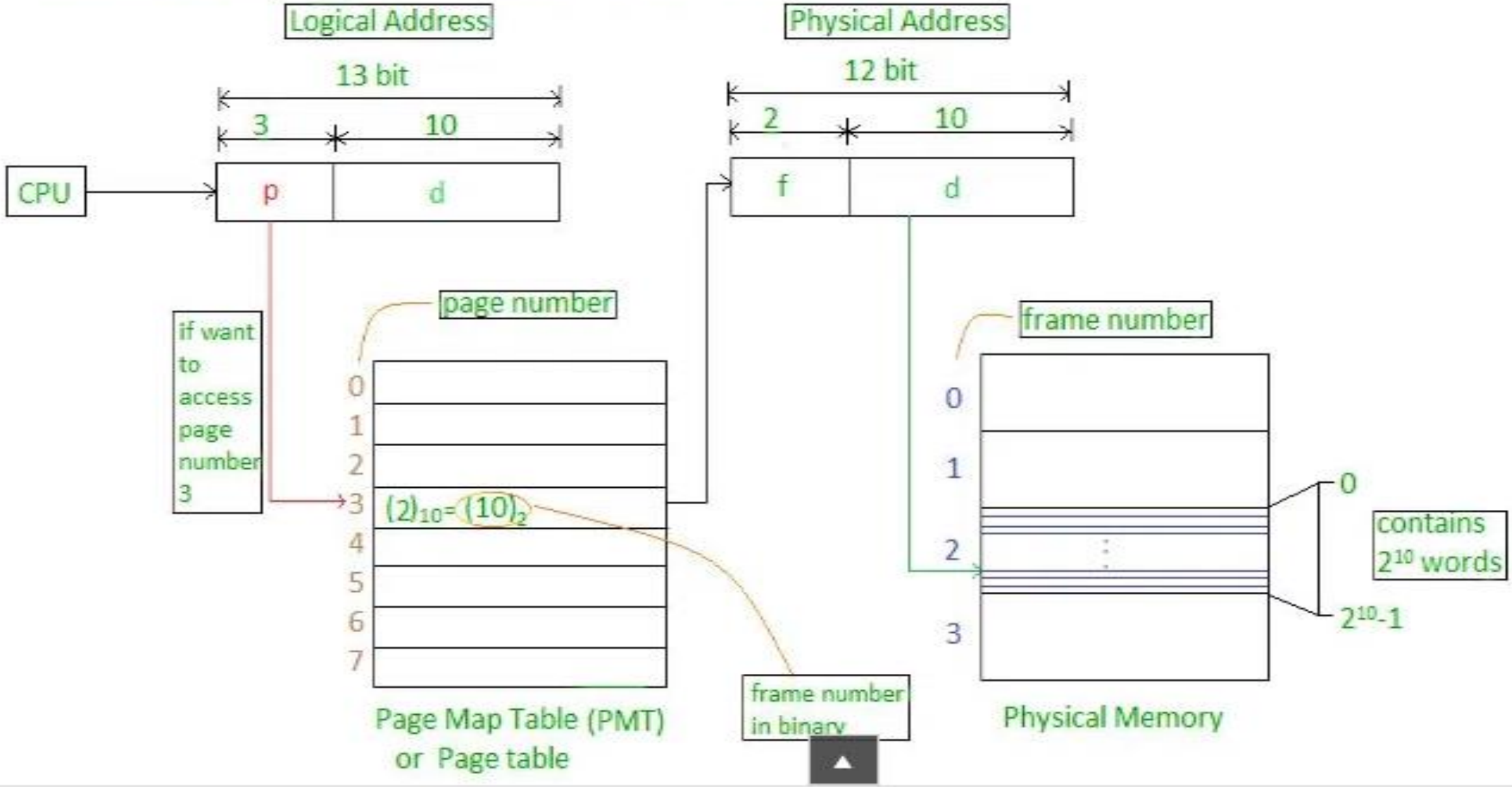
Paging in Hardware

- The paging unit translates linear addresses into physical ones.
- It checks 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.
- 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.
- The paging unit thinks of all RAM as partitioned into fixed-length page frames.
- Each page frame contains a page, i.e., 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.
- 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.
- In Intel processors, paging is enabled by setting the **PG flag** of the **cr0** register. When **PG = 0**, linear addresses are interpreted as physical addresses.


CPU
logical address
p
d
page number
frame number
TLB hit
physical address
f
d
TLB
p
TLB miss
f
page table
physical memory

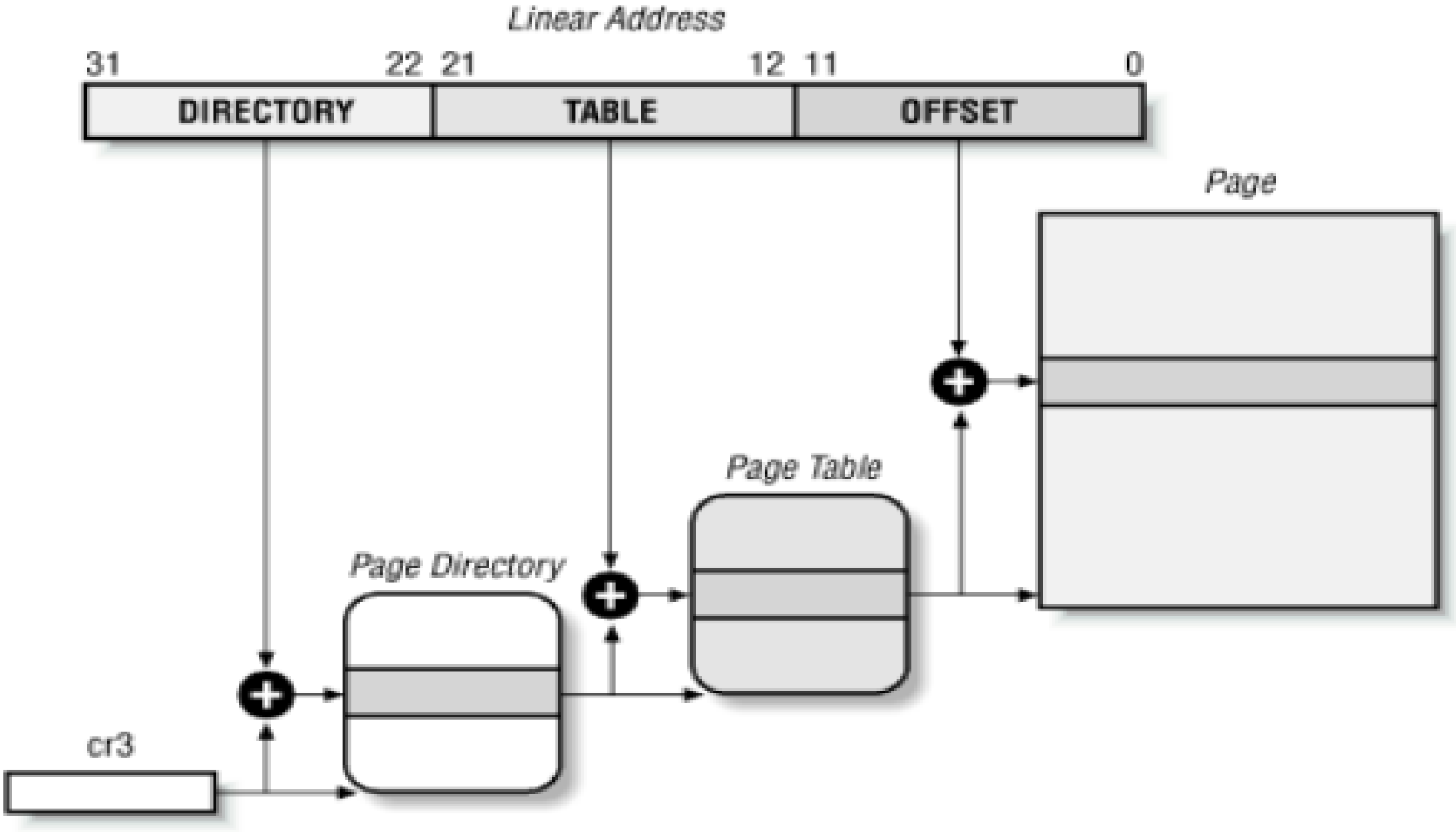
Number of frames = Physical Address Space / Frame size = 4 K / 1 K = 4 = 2^2

Number of pages = Logical Address Space / Page size = 8 K / 1 K = 8 = 2^3



Regular Paging

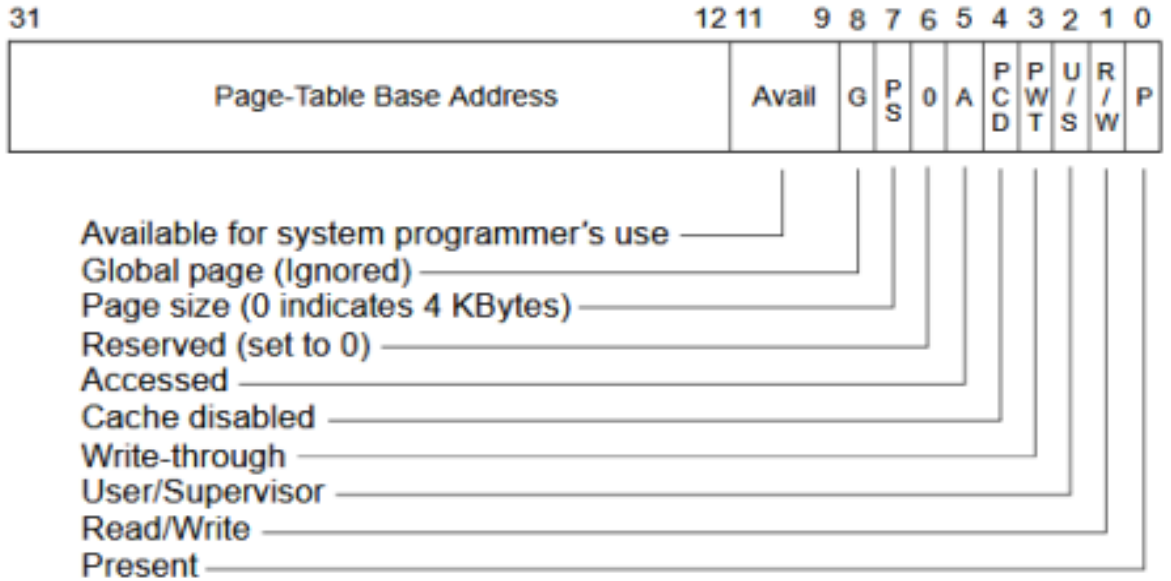
- Starting with the i80386, the paging unit of Intel processors handles 4 KB pages. The 32 bits of a linear address are divided into three fields:
  - Directory - The most significant 10 bits
  - Table - The intermediate 10 bits
  - Offset - The least significant 12 bits
- The translation of linear addresses is accomplished in two steps, each based on a type of translation table. The first translation table is called PageDirectory and the second is called Page Table.
- The physical address of the Page Directory in use is stored in the cr3 processor register.
- 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. Since it is 12 bits long, each page consists of 4096 bytes of data.


Linear Address
31
22
21
12
11
0
DIRECTORY
TABLE
OFFSET
Page
Page Table
Page Directory
cr3

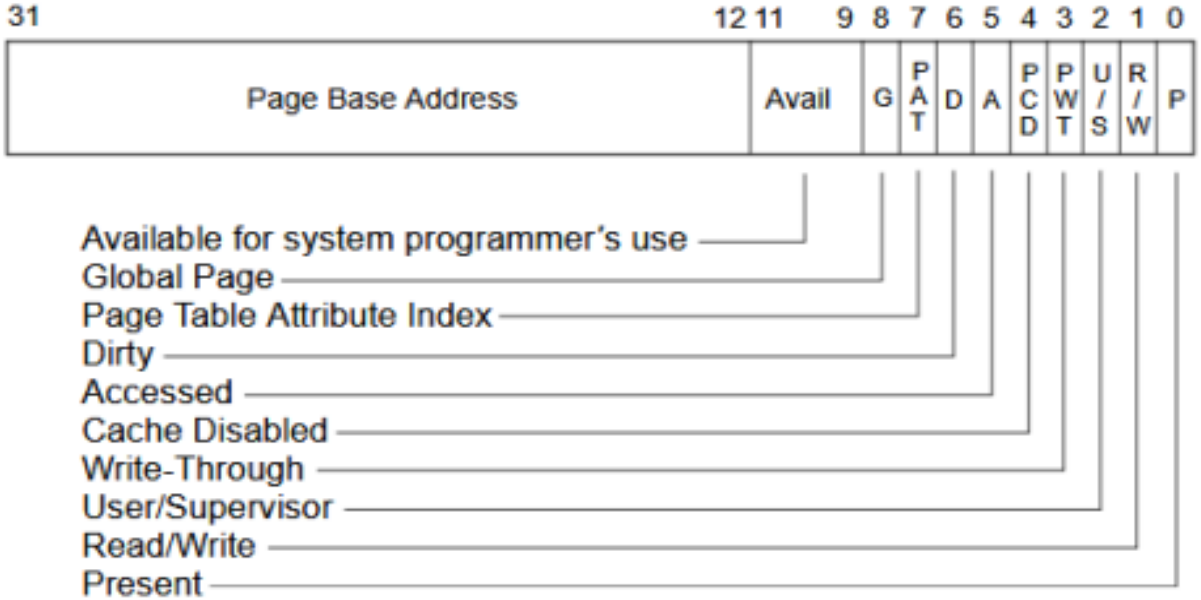
- Both the Directory and the Table fields are 10 bits long, so Page Directories and Page Tables can include up to 1024 entries. It follows that a Page Directory can address up to 1024 x 1024 x 4096=232 memory cells.
- The entries of Page Directories and Page Tables have the same structure. Each entry includes the following fields:
  - Present Flag - If it is set, the referred 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.
  - Field containing the 20 most significant bits of a page frame physical address - Since 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.
  - Accessed flag - Is 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 in the previous case, 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.
- Read/Write flag - Contains the access right (Read/Write or Read) of the page or of the Page TableRead/Write flag.
- User/Supervisor flag - Contains the privilege level required to access the page or Page Table.
- Two flags called PCD and PWT- Control the way the page or Page Table is handled by the hardware cache.
- Page Size flag - Applies only to Page Directory entries. If it is set, the entry refers to a 4 MB long page frame.
- 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 the cr2 processor register and generates the exception 14, that is, the "Page fault" exception.

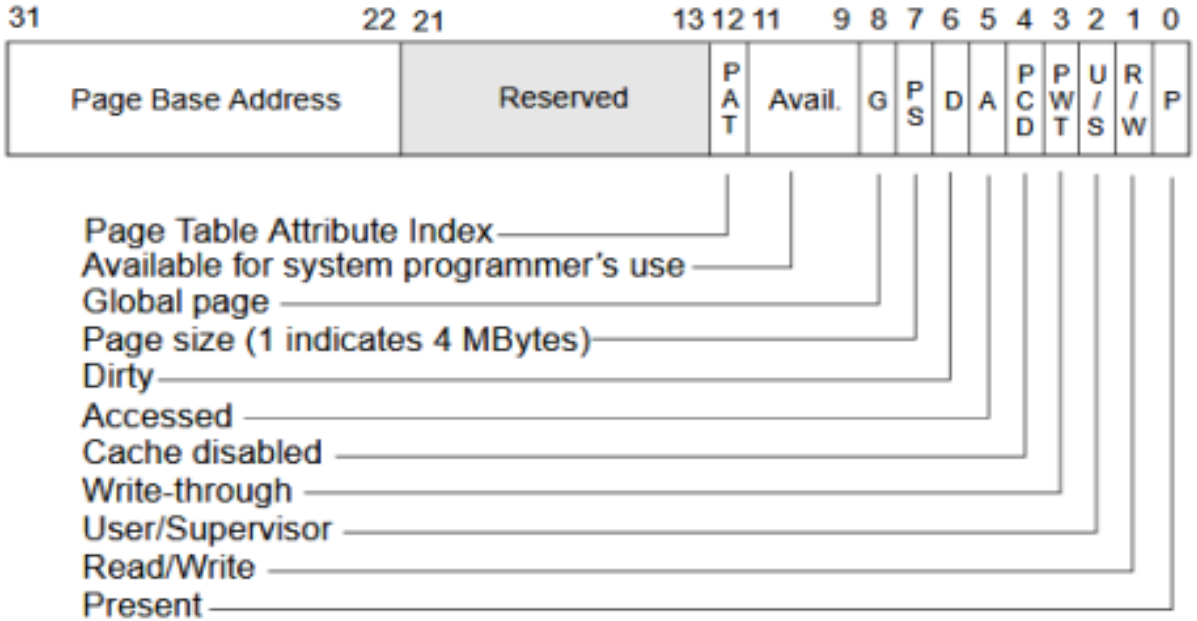
Page-Directory Entry (4-KByte Page Table)



Page-Table Entry (4-KByte Page)



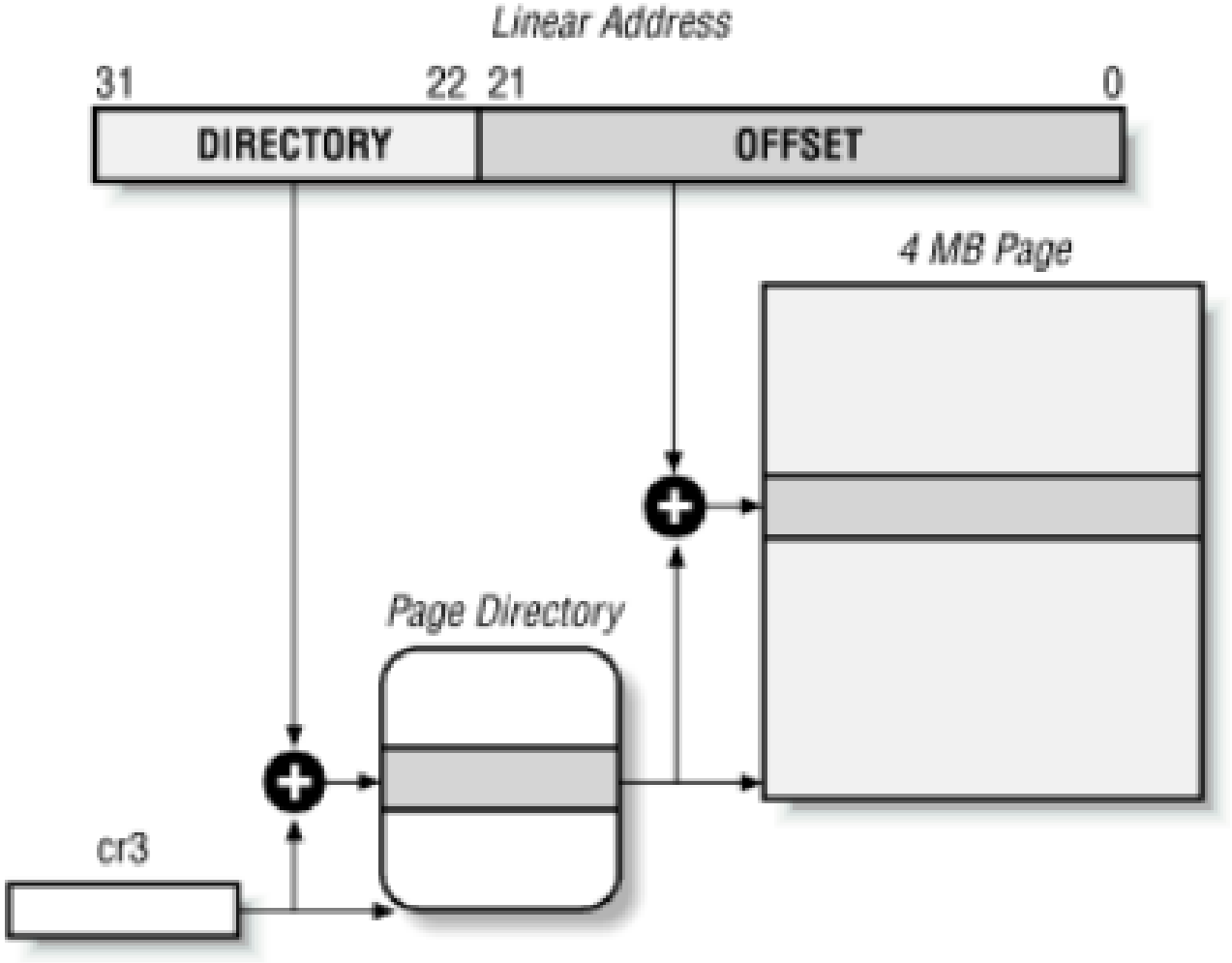
Page-Directory Entry (4-MByte Page)



Protected-mode memory management in x86 (source: Intel)

Extended Paging

- Starting with the Pentium model, Intel 80x86 microprocessors introduce extended paging , which allows page frames to be either 4 KB or 4 MB in size.



- 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 first 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 is used to translate large intervals of contiguous linear addresses into corresponding physical ones; in these cases, the kernel can do without intermediate Page Tables and thus save memory.

Hardware Protection Scheme

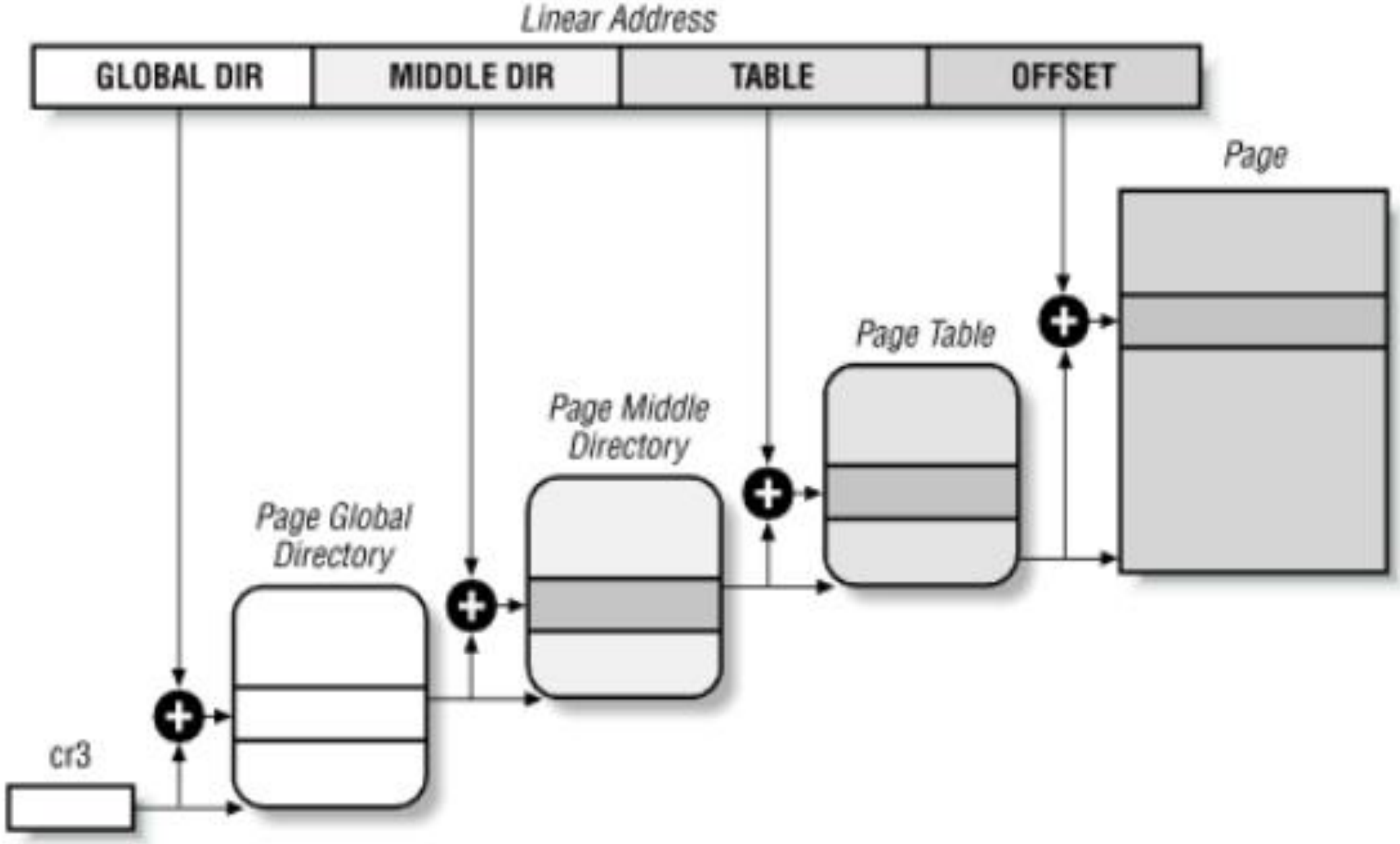
- The paging unit uses a different protection scheme from the segmentation unit.
- Here, only two privilege levels are associated with pages and Page Tables, because privileges are controlled by the User/Supervisor flag.
- When this flag is 0, the page can be addressed only when the CPL is less than 3.
- When the flag is 1, the page can always be addressed.
- Also here, only two types of access rights (Read, 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.

Three Level Paging

- Two-level paging is used by 32-bit microprocessors. Several microprocessors have adopted a 64-bit architecture.
- Let's choose 16 KB for the page size. Since 1 KB covers a range of 210 addresses, 16 KB covers 214 addresses, so the Offset field would be 14 bits. This leaves 50 bits of the linear address to be distributed between the Table and the Directory fields. If we now decide to reserve 25 bits for each of these two fields, this means that both the Page Directory and the Page Tables of a process would include 225 entries, that is, more than 32 million entries.
- Solution:-
- Page frames are 8 KB long, so the Offset field is 13 bits long.
- Only the least significant 43 bits of an address are used. (The most significant 21 bits are always set 0.)
- Three levels of page tables are introduced so that the remaining 30 bits of the address can be split into three 10-bit fields (see Figure 2-9 later in this chapter). So the Page Tables include 210 = 1024 entries as in the two-level paging schema examined previously.

Paging in Linux

- Linux adopted a three-level paging model so paging is feasible on 64-bit architectures.
- There are three types of paging tables:
  - Page Global Directory
  - Page Middle Directory
  - Page Table
- The Page Global Directory includes the addresses of several Page Middle Directories, which in turn include the addresses of several Page Tables. Each Page Table entry points to a page frame. The linear address is thus split into four parts.


Linear Address
GLOBAL DIR
MIDDLE DIR
TABLE
OFFSET
Page
Page Table
Page Middle Directory
Page Global Directory
cr3

- Linux 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, thus ensuring an efficient protection against addressing errors.
  - Distinguish pages, that is, groups of data, from page frames, that is, 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.

The Linear Address field

- The following macros simplify page table handling:
- PAGE_SHIFT - Specifies the length in bits of the Offset field; when applied to Pentium processors it yields the value 12. Since all the addresses in a page must fit in the Offset field, the size of a page on Intel 80x86 systems is 212 or the familiar 4096 bytes; the PAGE_SHIFT of 12 can thus be considered the logarithm base 2 of the total page size. This macro is used by PAGE_SIZE to return the size of the page. Finally, the PAGE_MASK macro is defined as the value 0xfffff000; it is used to mask all the bits of the Offset field.
- PMD_SHIFT
- Determines the number of bits in an address that are mapped by the second-level page table. It yields the value 22 (12 from Offset plus 10 from Table). The PMD_SIZE macro computes the size of the area mapped by a single entry of the Page Middle Directory, that is, of a Page Table. Thus, PMD_SIZE yields 222 or 4 MB. The PMD_MASK macro yields the value 0xffc00000; it is used to mask all the bits of the Offset and Table fields.

- PGDIR_SHIFT - Determines the logarithm of the size of the area a first-level page table can map. Since the Middle Directory field has length 0, this macro yields the same value yielded by PMD_SHIFT, which is 22. The PGDIR_SIZE macro computes the size of the area mapped by a single entry of the Page Global Directory, that is, of a Page Directory. PGDIR_SIZE therefore yields 4 MB. The PGDIR_MASK macro yields the value 0xffc00000, the same as PMD_MASK.

- PTRS_PER_PTE , PTRS_PER_PMD , and PTRS_PER_PGD - Compute the number of entries in the Page Table, Page Middle Directory, and Page 22Global Directory; they yield the values 1024, 1, and 1024, respectively