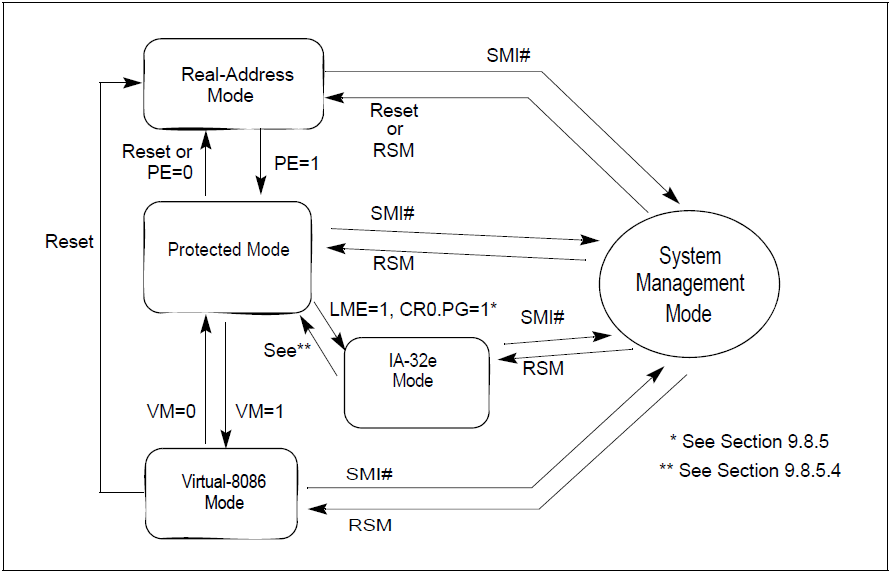
**Intel Architecture**

**Modes of Operation**

****

**Real Mode**

* Can only address 1MB of memory (0x0-0xFFFFF)
* All code can access any location in memory R/W

**Protected Mode**

* Can address 4GB of memory (0x0-0xFFFFFFFF)
* Each program can be assigned an isolated segment
* Provides four levels of privilege (rings 0-3)
* System architecture supports either direct physical addressing of memory or virtual memory (through paging). When physical addressing is used, a linear address is treated as a physical address.

**VM86 Mode**

* Compatibility feature to allow Real Mode applications to run within Protected Mode
* Each program gets a 1MB container and allows Protected Mode OS to hook certain operations (e.g. I/O access)

**System Management Mode (SMM)**

* Provides flat 4GB addressing similar to Real Mode
* Isolated address space independent of standard addressing
* Can only be entered by interrupt from chipset (SMI) and exited via RSM instruction
* Used by OEMs to develop custom power management firmware, hardware-specific features, patch around chipset bugs, etc.

**IA-32e (EM64T) Mode**

* Adds support for 64-bit address space
  + Compatibility Mode – 32-bit virtual addresses
  + 64-bit Mode (long mode) – native 64-bit virtual addresses

**VT Operating Mode**

* VMX root or host mode:
  + Virtual Machine Monitor (VMM) runs here
  + Full platform control with certain several event-blockings
  + Services guests’ requests (which cause VMexit to VMX host)
  + Configure guests’ operation before VMentry to guests
* VMX non-root or guest mode:
* User’s (guest’s) applications run here
* Do not have full platform control
* VMexit to host mode when configured to do so on certain platform access.

**System Registers**

In addition to the memory translation system registers (e.g.- GTDR, etc), the system architecture provides system flags in the EFLAGS register and several system registers:

* The system flags and IOPL field in the EFLAGS register control task and mode switching, interrupt handling, instruction tracing, and access rights.
* The control registers (CR0, CR2, CR3, and CR4) contain a variety of flags and data fields for controlling system-level operations. Other flags in these registers are used to indicate support for specific processor capabilities within the operating system.
* The debug registers allow the setting of breakpoints for use in debugging programs and systems software.
* Model Specific Registers (MSRs)

On systems that support IA-32e mode, the extended feature enable register (IA32\_EFER) is available. This model-specific register controls activation of IA-32e mode and other IA-32e mode operations. In addition, there are several model- specific registers that govern IA-32e mode instructions:

* IA32\_KernelGSbase — Used by SWAPGS instruction.
* IA32\_LSTAR — Used by SYSCALL instruction.
* IA32\_SYSCALL\_FLAG\_MASK — Used by SYSCALL instruction.
* IA32\_STAR\_CS — Used by SYSCALL and SYSRET instruction.

In 64-bit mode, the RFLAGS register expands to 64 bits with the upper 32 bits reserved. In IA-32e mode, the processor does not allow the VM bit to be set because virtual- 8086 mode is not supported (attempts to set the bit are ignored). Also, the processor will not set the NT bit.

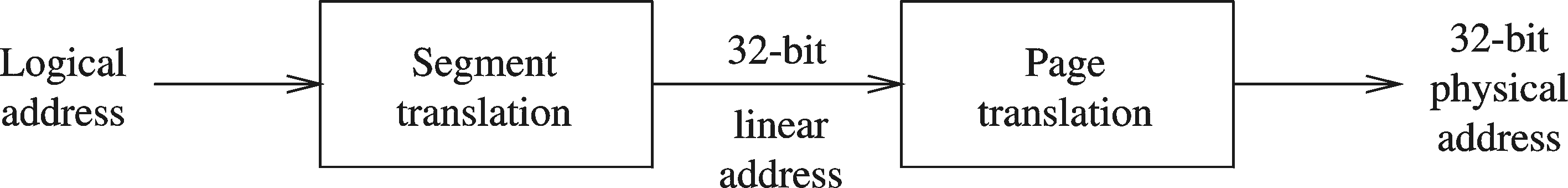
**General Registers**

* EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, EIP
* In 16 bit mode: CS, DS, ES, FS, GS, SS

**Address Translation**

Three kinds of addresses:

1. Logical (Segment, offset)
2. Linear (Flat virtual address)
3. Physical (address on pins after canonical address translation)



Logical address consists of

* 16-bit segment selector (CS, SS, DS, ES, FS, GS)
* Offset: Size depends on mode (16,32,64 bits).

Segment unit translates logical address to linear address using a segment descriptor table.

The *linear address* is 32 bits (called also a *virtual address*) in 32 bit mode, 64 bits in 64 bit mode.

The paging unit translates linear address to *physical address* (if PG bit is set) or it can be directly mapped

**Real Mode Address Translation**

A program can access up to six segments at any time

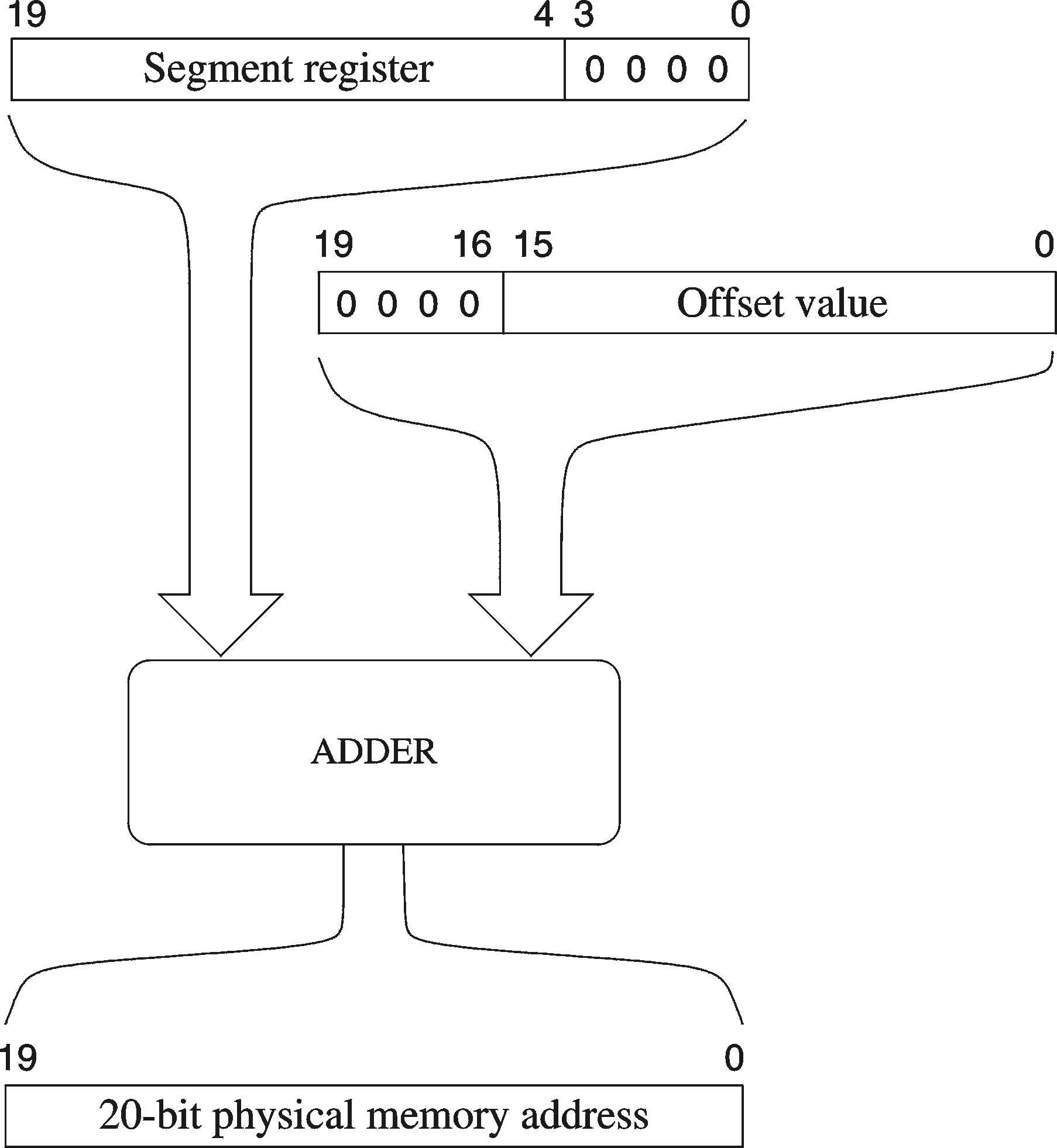
* Code segment
* Stack segment
* Data segment
* Extra segments (up to 3)

Each segment is 64 KB

Logical address

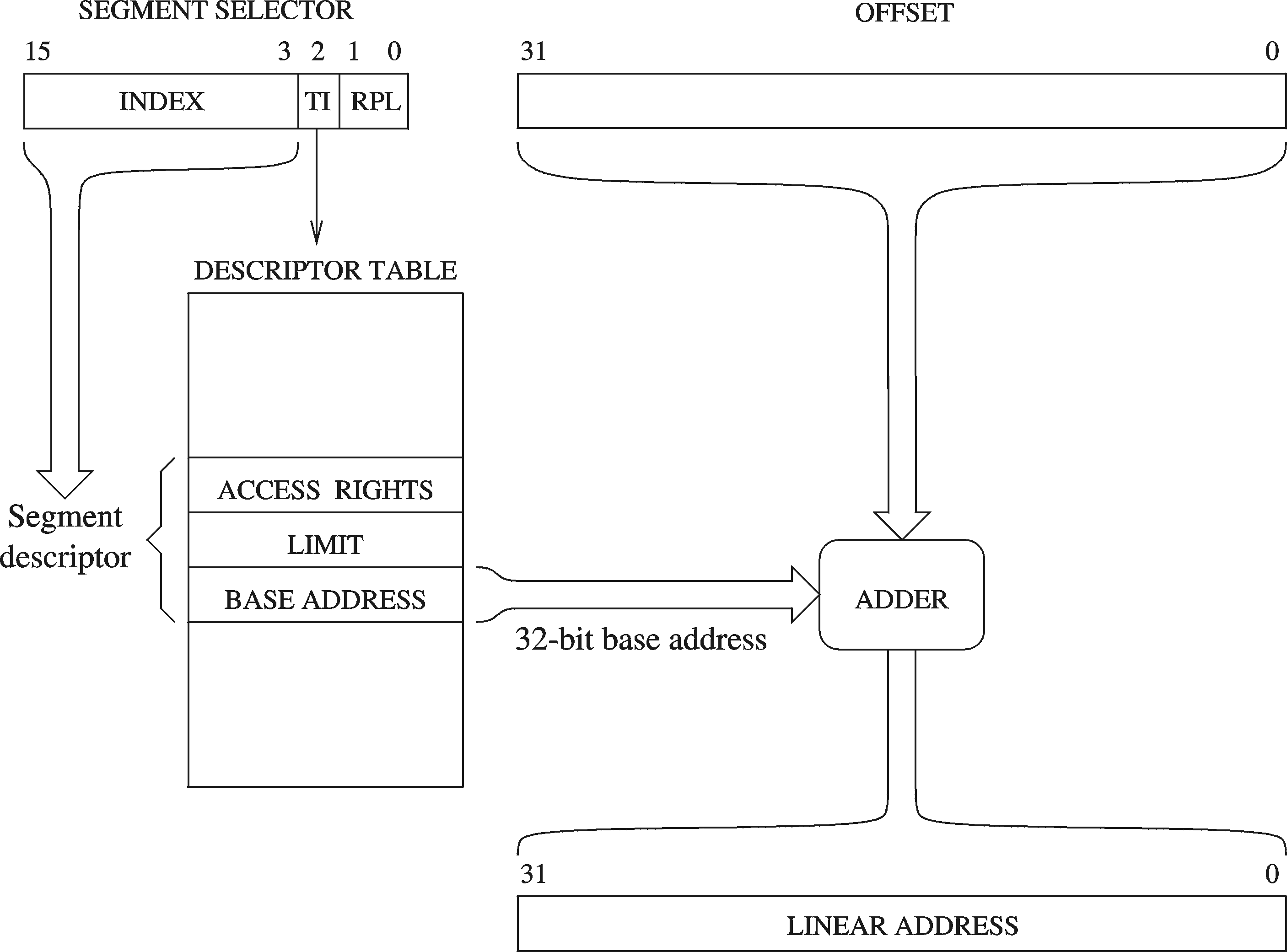
* Segment = 16 bits
* Offset = 16 bits

Linear (physical) address = 20 bits: (Segbase<<4)+Offset

****

****

**Protected Mode Address Translation**



Segment registers hold segment selectors.

The *segment selector* (16 bits) contains the following items:

* Index (Bits 3 through 15) — Selects one of 8192 descriptors in the GDT or LDT. The processor multiplies the index value by 8 (the number of bytes in a segment descriptor) and adds the result to the base address of the GDT or LDT (from the GDTR or LDTR register, respectively).
* TI (table indicator) flag (Bit 2) — Specifies the descriptor table to use: clearing this flag selects the GDT; setting this flag selects the current LDT.
* Requested Privilege Level (RPL) (Bits 0 and 1) — Specifies the privilege level of the selector. The privilege level can range from 0 to 3, with 0 being the most privileged level.





The *linear address* of the base of the GDT is contained in the **GDTR** register along with the table size. The *linear address* of the LDT is contained in the LDTR register which also has a segment selector. The GDTR and LDTR registers are 64-bits wide in both IA-32e sub-modes. Global and local descriptor tables are expanded in 64-bit mode to support 64-bit base addresses. In compatibility mode, descriptors are not expanded.

GDTR/IDTR format (48 bits) in protected mode, 80 bits in IA-32e mode

* 47:16 is segment base address
* 15:0 is segment limit



The first entry of the GDT is not used by the processor, this is the “null segment selector.”





Every *segment register* has a “visible” part and a “hidden” part.

To translate a logical address into a linear address, the processor does the following:

1. Uses the offset in the segment selector to locate the segment descriptor for the segment in the GDT or LDT and reads it into the processor. This step is needed only when a new segment selector is loaded into a segment register.
2. Examines the segment descriptor to check the access rights and range of the segment to insure that the segment is accessible and that the offset is within the limits of the segment.
3. Adds the base address (32 bits) of the segment from the segment descriptor to the offset (32 bits) to form a 32 bit linear address.

In IA-32e mode, an Intel 64 processor translates a logical address to a linear address. In 64-bit mode, the offset and base address of the segment are 64-bits instead of 32 bits. The linear address format is also 64 bits wide and is subject to the canonical form requirement.

Each code segment descriptor provides an L bit. This bit allows a code segment to execute 64-bit code or legacy 32-bit on a per code segment basis.

Besides code, data, and stack segments that make up the execution environment of a program or procedure, the architecture defines two system segments: the Task State Segment (“TSS”) and the LDT. The GDT is not considered a segment because it is not accessed by means of a segment selector and segment descriptor. TSSs and LDTs have segment descriptors defined for them.

The simplest memory model for protected mode is the basic “flat model:”

* Two segment descriptors must be created, one for referencing a code segment and one for referencing a data segment.
* Both segment descriptors have the same base address value of 0 and the same segment limit of 4 GBytes (in 32 bit mode).
* ROM (EPROM) is generally located at the top of the physical address space, because the processor begins execution at FFFF\_FFF0H.
* RAM (DRAM) is placed at the bottom of the address space because the initial base address for the DS data segment after reset initialization is 0.
* The protected flat model is similar to the basic flat model, except the segment limits are set to include only the range of addresses for which physical memory actually exists. A general-protection exception (#GP) is then generated on any attempt to access nonexistent memory. This model provides a minimum level of hardware protection against some kinds of program bugs.

Two kinds of load instructions are provided for loading the segment registers:

1. Direct load instructions such as the MOV, POP, LDS, LES, LSS, LGS, and LFS instructions. These instructions explicitly reference the segment registers. The MOV instruction can also be used to store visible part of a segment register in a general-purpose register.
2. Implied load instructions such as the far pointer versions of the CALL, JMP, and RET instructions, the SYSENTER and SYSEXIT instructions, and the IRET, INTn, INTO and INT3 instructions. These instructions change the contents of the CS register (and sometimes other segment registers) as an incidental part of their operation.

The privileged instructions used in memory address translation are:

LGDT — Load GDT register.

LLDT — Load LDT register.

LTR — Load task register.

LIDT — Load IDT register.

MOV (control registers) — Load and store control registers.

LMSW — Load machine status word.

CLTS — Clear task-switched flag in register CR0.

MOV (debug registers) — Load and store debug registers.

INVD — Invalidate cache, without writeback.

WBINVD — Invalidate cache, with writeback.

INVLPG — Invalidate TLB entry.

HLT — Halt processor.

RDMSR — Read Model-Specific Registers.

WRMSR — Write Model-Specific Registers.

RDPMC — Read Performance-Monitoring Counter.

RDTSC — Read Time-Stamp Counter.

**Privileges (all modes)**

CPL – Current Privilege Level

* Usually equal to the DPL of the currently executing code segment

DPL – Descriptor Privilege Level

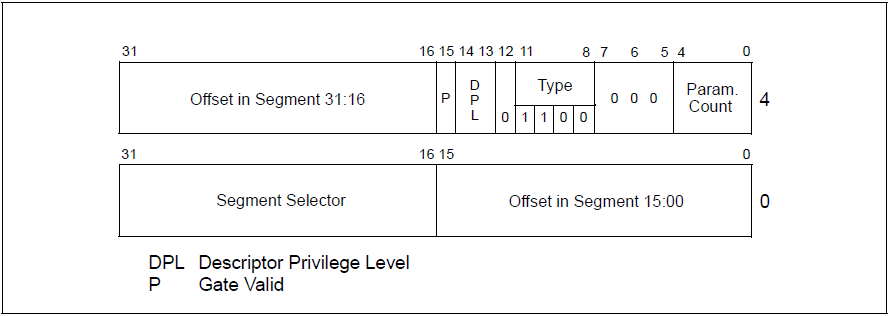
* Defined in the descriptor for the segment

RPL – Requestor Privilege Level

* Written in the segment register
* Can be used to lower privilege level

IOPL – I/O Privilege Level

* Defined in EFLAGS, stored in TSS – defines privilege level required to execute IN, INS, OUT, OUTS, CLI, and STI
* Conforming vs. Non-Conforming code segments
  + Defined in C bit in segment descriptor
  + Non-Conforming – can only be jumped/called to by code with CPL == DPL
  + Conforming – can be jumped to/called by code with CPL of equal or lesser privilege (CPL >= DPL). Privilege of code segment is then lowered to remain the same as calling code segment.



**IA-32e protected mode translation**

* In compatibility mode, segmentation functions just as it does using legacy 16-bit or 32-bit protected mode semantics.
* In 64-bit mode, segmentation is generally disabled, creating a flat 64-bit linear-address space. Note that the processor does not perform segment limit checks at runtime in 64-bit mode.

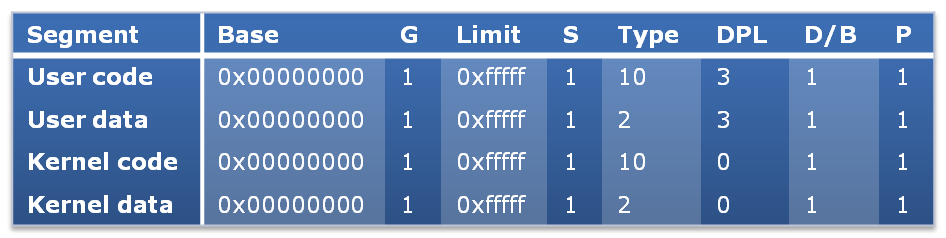
Because ES, DS, and SS segment registers are not used in 64-bit mode, their fields (base, limit, and attribute) in segment descriptor registers are ignored. Some forms of segment load instructions are also invalid (for example, LDS, POP ES). Address calculations that reference the ES, DS, or SS segments are treated as if the segment base is zero.

The processor checks that all linear-address references are in *canonical form* instead of performing limit checks. Mode switching does not change the contents of the segment registers or the associated descriptor registers. These registers are also not changed during 64-bit mode execution, unless explicit segment loads are performed.

****

****

**Linux Segments**



**Tasks**

All program execution in protected mode happens within the context of a task (called the current task). A task is made up of two parts: a task execution space and a task-state segment (TSS). The task execution space consists of a code segment, a stack segment, and one or more data segments. The TSS defines the state of the execution environment for a task. If an operating system uses the processor’s privilege-level protection mechanism, the task execution space provides a separate stack for each privilege level.

The *task register* (“TR”) holds the 16-bit segment selector, base address (32 bits in protected mode; 64 bits in IA-32e mode), segment limit, and descriptor attributes for the TSS of the current task. The selector references the TSS descriptor in the GDT. The base address specifies the linear address of byte 0 of the TSS; the segment limit specifies the number of bytes in the TSS. On power up or reset of the processor, the base address is set to the default value of 0 and the limit is set to 0FFFFH.

When a task switch occurs, the task register is automatically loaded with the segment selector and descriptor for the TSS for the new task. The contents of the task register are not automatically saved prior to writing the new TSS information into the register.

In multitasking systems, the TSS also provides a mechanism for linking tasks. It includes: the state of general-purpose registers and segment registers, the EFLAGS register, the EIP register, and the segment selectors with stack pointers for three stack segments (one stack for each privilege level). The TSS also includes the segment selector for the LDT associated with the task and the base address of the paging structure hierarchy. The segment selector for the base address of the LDT must be in the GDT.

The simplest method for switching to a task is to make a call or jump to the new task. Here, the segment selector for the TSS of the new task is given in the CALL or JMP instruction.

In switching tasks, the processor performs the following actions:

1. Stores the state of the current task in the current TSS.

2. Loads the task register with the segment selector for the new task.

3. Accesses the new TSS through the task segment descriptor in the GDT.

4. Loads the state of the new task from the new TSS into the general-purpose registers, the segment registers, the LDTR, control register CR3 (base address of the paging-structure hierarchy), the EFLAGS register, and the EIP register.

5. Begins execution of the new task.

A task can also be accessed through a task gate. A task gate is similar to a call gate, except that it provides access (through a segment selector) to a TSS rather than a code segment. Hardware task switches are not supported in IA-32e mode. However, TSSs continue to exist.

A 64-bit TSS holds the following information that is important to 64-bit operation:

* Stack pointer addresses for each privilege level
* Pointer addresses for the interrupt stack table
* Offset address of the IO-permission bitmap (from the TSS base)

Each task is created by allocating a Task State Segment in the GDT

* Current TSS pointed to by TR register
* Task switch triggered with far jump or CALL with CS set to TSS
* Hardware discards offset, saves CPU state into current TSS and loads state from selected TSS
* Doesn’t work well with paging since all TSS must have same linear-to-physical mapping in order to load new TSS linear values
* Requires entire task to be loaded into memory!
* Linux sets up one TSS for each logical CPU
* Needed for several reasons, even though hardware task switching is not used with paging:
* Switching from user (ring 3) mode to kernel (ring 0) mode requires looking up the stack for kernel mode from the current TSS
* User process attempting to execute I/O instructions may have to check the I/O permission bitmap in the current TSS if the current EFLAGS IOPL is not 3

**Switching Privilege levels**

There are three ways to switch privilege levels programmatically:

1. Call Gate – special descriptor, hardware performs code segment switch, stack switch, copies stack arguments to new stack, and changes CPL. Can access using far call/jmp/ret.
2. Soft-interrupt (INT n) – invokes interrupt gate handler for the specified vector, which can then examine source for parameters
3. SYSENTER/SYSEXIT – loads CS, EIP, SS, and ESP from special architectural MSRs

The architecture provides a set of special descriptors called gates (call gates, interrupt gates, trap gates, and task gates). These provide protected gateways to system procedures and handlers that may operate at a different privilege level than application programs and most procedures.

To access a procedure through a call gate, the calling procedure supplies the selector for the call gate. The processor then performs an access rights check on the call gate, comparing the CPL with the privilege level of the call gate and the destination code segment pointed to by the call gate. If access to the destination code segment is allowed, the processor gets the segment selector for the destination code segment and an offset into that code segment from the call gate. If the call requires a change in privilege level, the processor also switches to the stack for the targeted privilege level. The segment selector for the new stack is obtained from the TSS for the currently running task. Gates also facilitate transitions between 16-bit and 32-bit code segments, and vice versa.

In IA-32e mode, the following descriptors are 16-byte descriptors (expanded to allow a 64-bit base): LDT descriptors, 64-bit TSSs, call gates, interrupt gates, and trap gates. Call gates facilitate transitions between 64-bit mode and compatibility mode. Task gates are not supported in IA-32e mode. On privilege level changes, stack segment selectors are not read from the TSS. Instead, they are set to NULL.

**Switching Privilege changes not caused programmatically**

1. Special cycles

* TXT, aborts

1. Interrupts – occur asynchronously, generated by “external” events.

* E.g. network card has a new incoming packet, hard driver controller has data ready, etc.
* Includes maskable hardware interrupts, non-maskable hardware interrupts, and software interrupts using INT n instruction.

1. Exceptions – occur as a result of software execution. Exceptions are divided into faults, traps, and aborts

* Faults – exception that can be corrected by handler and restarted
  + Faulting instruction does not retire, return address set to faulting instruction
  + E.g. #PF – OS page fault handler pages in non-present page and restarts instruction
* Traps – exception reported immediately \*after\* trapping instruction
  + E.g. #OF – Overflow occurs and trap reported for handling after retirement.
* Aborts – non-recoverable error conditions
  + E.g. #MC – machine check

Each interrupt, trap and exception has an integer “vector”, i (0<=i<256), that is an index into the Interrupt Descriptor Table (“IDT”) whose base *linear address* in IDTR; it has no segment selector. Each entry in the IDT is a descriptor similar to entries in the GDT. The first 32 entries are fixed or reserved and the remaining are configurable by the OS. In IA-32e mode, interrupt descriptors are expanded to 16 bytes to support 64-bit base addresses and the IDTR register is also 64-bits. Task gates are not supported.

Descriptors are of three types:

* Task gate
  + Replaces the current TSS with TSS supplied in descriptor
  + Effectively uses a hardware task to respond to vector
* Interrupt gate
  + Descriptor includes segment selector and offset of interrupt handler
  + Clears IF flag, disabling further maskable interrupts
* Trap gate
  + Descriptor includes segment selector and offset of trap handler
  + Does not modify IF, thereby allowing further maskable interrupts

Linux generally follows the SMP model (Symmetric Multi-Processing)

* All cores treated equally when it comes to interrupt routing
* TPR set to the same value for all

Linux uses slightly different terminology from Intel:

* Interrupt gate – kernel mode only interrupt gate (DPL=0)
* System gate – user mode trap gate (DPL=3), e.g. vectors 4, 5, and 128 to support INTO, BOUND, and INT 0x80
* System interrupt gate – user mode interrupt gate (DPL=3), e.g. vector 3 to support INT3
* Trap gate – user mode trap gate (DPL=3), e.g. most exception handlers
* Task gate – kernel mode task gate (DPL=0), e.g. to support double fault exception handler

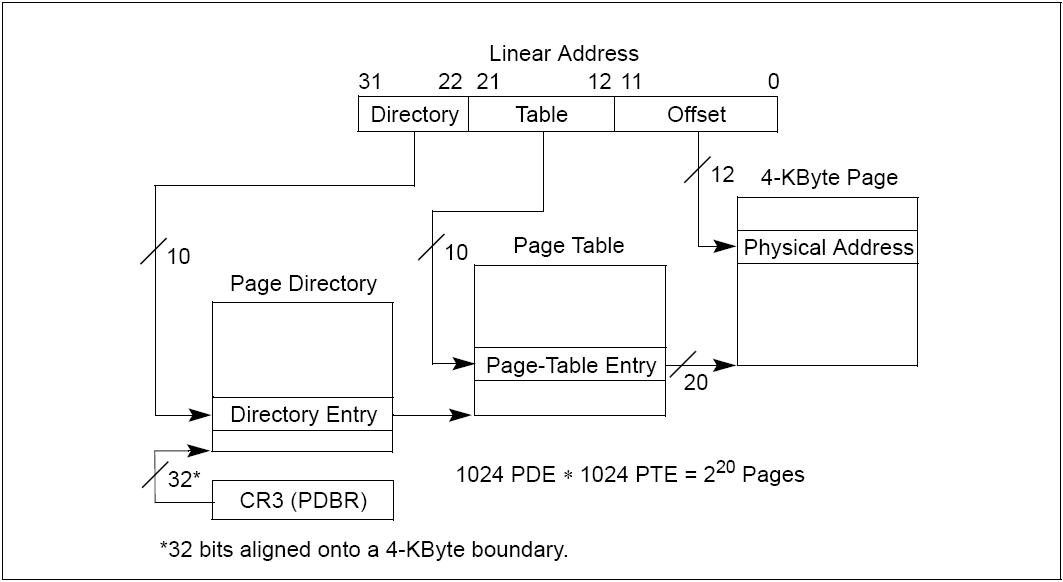
**IDT in Linux**

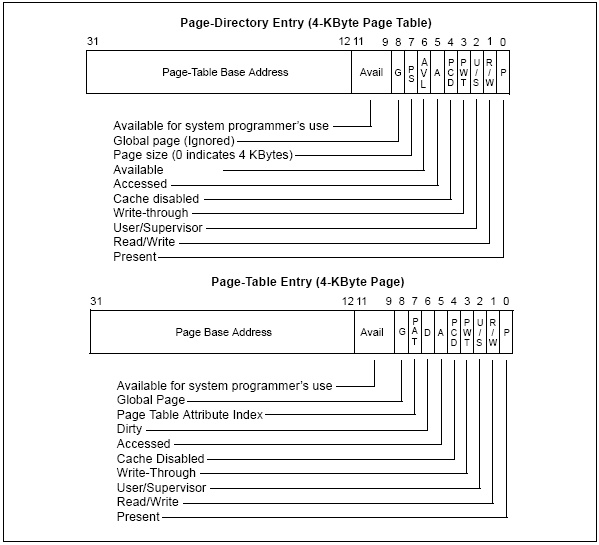
set\_trap\_gate(0, &divide\_error);

set\_intr\_gate(1, &debug);

set\_intr\_gate(2, &nmi);

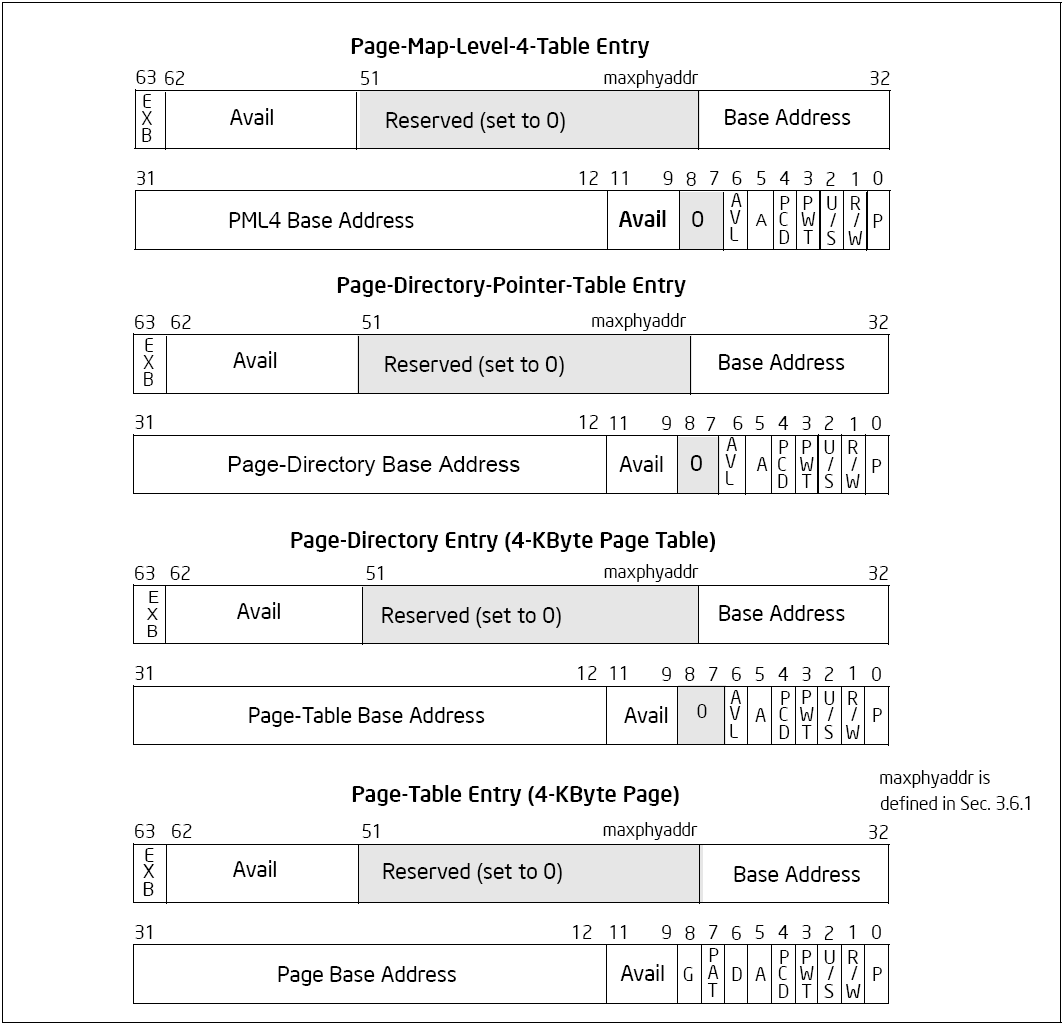
**Paging Structures**





In IA-32e mode, physical memory pages are managed by a set of system data structures. In compatibility mode and 64-bit mode, four levels of system data structures are used. These include:

* The page map level 4 (PML4) — An entry in table contains the physical address of the base of a page directory pointer table, access rights, and memory management information. The base physical address of the PML4 is stored in CR3.
* A set of page directory pointer tables — An entry contains the physical address of the base of a page directory table, access rights, and memory management information.
* Sets of page directories — An entry contains the physical address of the base of a page table, access rights, and memory management information.
* Sets of page tables — An entry contains the physical address of a page frame, access rights, and memory management information.



**I/O**

**Interrupt Controllers**

Programmable Interrupt Controller (PIC) is chip between devices and CPU. PIC translates IRQ to interrupt vector

* Raises interrupt to CPU
* Vector available in register
* Waits for ack from CPU

Other interrupts may be pending

* Possible to “mask” interrupts at PIC or CPU
* Fixed number of wires in from devices
  + IRQs: Interrupt ReQuest lines
* Single wire to CPU + some registers

*Advanced PIC (APIC)* for SMP systems

* Interrupts “routed” to CPU over system bus
* IPI: inter-processor interrupt
* Local APICs “frontend” IO-APIC
* Devices connect to front-end IO-APIC
* IO-APIC communicates (over bus) with Local APIC
* Interrupt routing
* Allows broadcast or selective routing of interrupts
* Need to distribute interrupt handling load
* Routes to lowest priority process
  + Special register: Task Priority Register (TPR)
* Arbitrates (round-robin) if equal priority

**Legacy I/O**

* Use IN, INS, OUT, OUTS instructions
* Separate 16-bit I/O address space to read/write I/O ports
* To use these instructions your CPL must be equal or higher privilege to IOPL, or the port addressed in the instruction must be 1 in the I/O permission bitmap of the TSS

**Memory-mapped I/O (MMIO)**

* Accessed using regular memory operations
* Portions of CPU memory map are mapped to chipset
* (Actually, everything that isn’t backed by DRAM is sent to chipset)
* To access you must have permissions via segmentation and paging just as a memory access would.

****

**System Flags**

The system flags and IOPL field of the EFLAGS register control I/O, maskable hardware interrupts, debugging, task switching, and the virtual-8086 mode (see Figure 2-4). Only privileged code (typically operating system or executive code) should be allowed to modify these bits.

The system flags and IOPL are:

* TF, Trap (bit 8) — Set to enable single-step mode for debugging; clear to disable single-step mode. In single-step mode, the processor generates a debug exception after each instruction. This allows the execution state of a program to be inspected after each instruction. If an application program sets the TF flag using a POPF, POPFD, or IRET instruction, a debug exception is generated after the instruction that follows the POPF, POPFD, or IRET.
* IF Interrupt enable (bit 9) — Controls the response of the processor to maskable hardware interrupt requests. The flag is set to respond to maskable hardware interrupts; cleared to inhibit maskable hardware interrupts. The IF flag does not affect the generation of exceptions or nonmaskable interrupts (NMI interrupts). The CPL, IOPL, and the state of the VME flag in control register CR4 determine whether the IF flag can be modified by the CLI, STI, POPF, POPFD, and IRET.
* IOPL, I/O privilege level field (bits 12 and 13) — Indicates the I/O privilege level (IOPL) of the currently running program or task. The CPL of the currently running program or task must be less than or equal to the IOPL to access the I/O address space. This field can only be modified by the POPF and IRET instructions when operating at a CPL of 0. The IOPL is also one of the mechanisms that controls the modification of the IF flag and the handling of interrupts in virtual-8086 mode when virtual mode extensions are in effect (when CR4.VME = 1).
* NT, Nested task (bit 14) — Controls the chaining of interrupted and called tasks. The processor sets this flag on calls to a task initiated with a CALL instruction, an interrupt, or an exception. It examines and modifies this flag on returns from a task initiated with the IRET instruction. The flag can be explicitly set or cleared with the POPF/POPFD instructions.
* RF, Resume (bit 16) — Controls the processor’s response to instruction-break- point conditions. When set, this flag temporarily disables debug exceptions (#DB) from being generated for instruction breakpoints (although other exception conditions can cause an exception to be generated). When clear, instruction breakpoints will generate debug exceptions. The primary function of the RF flag is to allow the restarting of an instruction following a debug exception that was caused by an instruction breakpoint condition. Here, debug software must set this flag in the EFLAGS image on the stack just prior to returning to the interrupted program with IRETD (to prevent the instruction breakpoint from causing another debug exception). The processor then automatically clears this flag after the instruction returned to has been successfully executed, enabling instruction breakpoint faults again.
* VM, Virtual-8086 mode (bit 17) — Set to enable virtual-8086 mode; clear to return to protected mode.
* AC, Alignment check (bit 18) — Set this flag and the AM flag in control register CR0 to enable alignment checking of memory references; clear the AC flag and/or the AM flag to disable alignment checking. An alignment-check exception is generated when reference is made to an unaligned operand, such as a word at an odd byte address or a doubleword at an address which is not an integral multiple of four. Alignment-check exceptions are generated only in user mode (privilege level 3). Memory references that default to privilege level 0, such as segment descriptor loads, do not generate this exception even when caused by instructions executed in user-mode. The alignment-check exception can be used to check alignment of data.
* VIF, Virtual Interrupt (bit 19) — Contains a virtual image of the IF flag. This flag is used in conjunction with the VIP flag. The processor only recognizes the VIF flag when either the VME flag or the PVI flag in control register CR4 is set and the IOPL is less than 3. (The VME flag enables the virtual-8086 mode extensions; the PVI flag enables the protected-mode virtual interrupts.)
* Virtual interrupt pending (bit 20) — Set by software to indicate that an interrupt is pending; cleared to indicate that no interrupt is pending. This flag is used in conjunction with the VIF flag. The processor reads this flag but never modifies it. The processor only recognizes the VIP flag when either the VME flag or the PVI flag in control register CR4 is set and the IOPL is less than 3. The VME flag enables the virtual-8086 mode extensions; the PVI flag enables the protected-mode virtual interrupts.
* Identification (bit 21). — The ability of a program or procedure to set or clear this flag indicates support for the CPUID instruction.

In IA-32e mode, the SYSCALL/SYSRET instructions have a programmable method of specifying which bits are cleared in RFLAGS/EFLAGS. These instructions save and restore EFLAGS/RFLAGS.

**Control Register Formats**

Control registers (CR0, CR1, CR2, CR3, and CR4) determine operating mode of the processor and the characteristics of the currently executing task. These registers are 32 bits in all 32-bit modes and compatibility mode. In 64-bit mode, control registers are expanded to 64 bits.

The MOV CRn instructions are used to manipulate the register bits. Operand-size prefixes for these instructions are ignored. The following is also true:

* Bits 63:32 of CR0 and CR4 are reserved and must be written with zeros. Writing a nonzero value to any of the upper 32 bits results in a general-protection exception, #GP(0).
* All 64 bits of CR2 are writable by software.
* Bits 51:40 of CR3 are reserved and must be 0.

The MOV CRn instructions do not check that addresses written to CR2 and CR3 are within the linear-address or physical address limitations of the implementation.

* CR0 — Contains system control flags that control operating mode and states of the processor.
* CR1 — Reserved.
* CR2 — Contains the page-fault linear address (the linear address that caused a page fault).
* CR3 — Contains the physical address of the base of the paging-structure hierarchy and two flags (PCD and PWT). Only the most-significant bits (less the lower 12 bits) of the base address are specified; the lower 12 bits of the address are assumed to be 0. The first paging structure must thus be aligned to a page (4-KByte) boundary. The PCD and PWT flags control caching of that paging structure in the processor’s internal data caches (they do not control TLB caching of page-directory information). When using the physical address extension, the CR3 register contains the base address of the page-directory-pointer table In IA-32e mode, the CR3 register contains the base address of the PML4 table.
* CR4 — Contains a group of flags that enable several architectural extensions, and indicate operating system or executive support for specific processor capabilities. The control registers can be read and loaded (or modified) using the move- to-or-from-control-registers forms of the MOV instruction. In protected mode, the MOV instructions allow the control registers to be read or loaded (at privilege level 0 only). This restriction means that application programs or operating- system procedures (running at privilege levels 1, 2, or 3) are prevented from reading or loading the control registers.
* CR8 — Provides read and write access to the Task Priority Register (TPR). It specifies the priority threshold value that operating systems use to control the priority class of external interrupts allowed to interrupt the processor. This register is available only in 64-bit mode. However, interrupt filtering continues to apply in compatibility mode.

When loading a control register, reserved bits should always be set to the values previously read. The flags in control registers are:

* PG, Paging (bit 31 of CR0) —Enables paging when set; disables paging when clear. When paging is disabled, all linear addresses are treated as physical addresses. The PG flag has no effect if the PE flag (bit 0 of register CR0) is not also set; setting the PG flag when the PE flag is clear causes a general- protection exception (#GP). On Intel 64 processors, enabling and disabling IA-32e mode operation also requires modifying CR0.PG.
* CD, Cache Disable (bit 30 of CR0) — When the CD and NW flags are clear, caching of memory locations for the whole of physical memory in the processor’s internal (and external) caches is enabled. When the CD flag is set, caching is restricted as described in Table 11-5. To prevent the processor from accessing and updating its caches, the CD flag must be set and the caches must be invalidated so that no cache hits can occur.
* NW, Not Write-through (bit 29 of CR0) — When the NW and CD flags are clear, write-back (for Pentium 4, Intel Xeon, P6 family, and Pentium processors) or write-through (for Intel486 processors) is enabled for writes that hit the cache and invalidation cycles are enabled. See Table 11-5 for detailed information about the affect of the NW flag on caching for other settings of the CD and NW flags.
* AM, Alignment Mask (bit 18 of CR0) — Enables automatic alignment checking when set; disables alignment checking when clear. Alignment checking is performed only when the AM flag is set, the AC flag in the EFLAGS register is set, CPL is 3, and the processor is operating in either protected or virtual- 8086 mode.
* WP, Write Protect (bit 16 of CR0) — When set, inhibits supervisor-level procedures from writing into read-only pages; when clear, allows supervisor-level procedures to write into read-only pages (regardless of the U/S bit setting; see Section 4.1.3 and Section 4.6). This flag facilitates implementation of the copy-on-write method of creating a new process (forking) used by operating systems such as UNIX.
* NE, Numeric Error (bit 5 of CR0) — Enables the native (internal) mechanism for reporting x87 FPU errors when set; enables the PC-style x87 FPU error reporting mechanism when clear. When the NE flag is clear and the IGNNE# input is asserted, x87 FPU errors are ignored. When the NE flag is clear and the IGNNE# input is deasserted, an unmasked x87 FPU error causes the processor to assert the FERR# pin to generate an external interrupt and to stop instruction execution immediately before executing the next waiting floating-point instruction or WAIT/FWAIT instruction.
* ET, Extension Type (bit 4 of CR0) — Reserved in the Pentium 4, Intel Xeon, P6 family, and Pentium processors. In the Pentium 4, Intel Xeon, and P6 family processors, this flag is hardcoded to 1. In the Intel386 and Intel486 processors, this flag indicates support of Intel 387 DX math coprocessor instructions when set.
* TS, Task Switched (bit 3 of CR0) — Allows the saving of the x87 FPU/MMX/SSE/SSE2/SSE3/SSSE3/SSE4 context on a task switch to be delayed until an x87 FPU/MMX/SSE/SSE2/SSE3/SSSE3/SSE4 instruction is actually executed by the new task. The processor sets this flag on every task switch and tests it when executing x87
* EM, Emulation (bit 2 of CR0) — Indicates that the processor does not have an internal or external x87 FPU when set; indicates an x87 FPU is present when clear. This flag also affects the execution of MMX/SSE/SSE2/SSE3/SSSE3/SSE4 instructions.
* MP, Monitor Coprocessor (bit 1 of CR0). — Controls the interaction of the WAIT (or FWAIT) instruction with the TS flag (bit 3 of CR0). If the MP flag is set, a WAIT instruction generates a device-not-available exception (#NM) if the TS flag is also set. If the MP flag is clear, the WAIT instruction ignores the setting of the TS flag. Table 9-2 shows the recommended setting of this flag, depending on the IA-32 processor and x87 FPU or math coprocessor present in the system. Table 2-1 shows the interaction of the MP, EM, and TS flags.
* PE, Protection Enable (bit 0 of CR0) — Enables protected mode when set; enables real-address mode when clear. This flag does not enable paging directly. It only enables segment-level protection. To enable paging, both the PE and PG flags must be set.
* PCD, Page-level Cache Disable (bit 4 of CR3) — Controls the memory type used to access the first paging structure of the current paging-structure hierarchy. This bit is not used if paging is disabled, with PAE paging, or with IA-32e paging if CR4.PCIDE=1.
* PWT, Page-level Write-Through (bit 3 of CR3) — Controls the memory type used to access the first paging structure of the current paging-structure hier- archy. This bit is not used if paging is disabled, with PAE paging, or with IA-32e paging if CR4.PCIDE=1.
* VME, Virtual-8086 Mode Extensions (bit 0 of CR4) — Enables interrupt and exception handling extensions in virtual-8086 mode when set; disables the extensions when clear. It also provides hardware support for a virtual interrupt flag (VIF) to improve reliability of running 8086 programs in multitasking and multiple-processor environments.
* PVI, Protected-Mode Virtual Interrupts (bit 1 of CR4) — Enables hardware support for a virtual interrupt flag (VIF) in protected mode when set; disables the VIF flag in protected mode when clear.
* TSD, Time Stamp Disable (bit 2 of CR4) — Restricts the execution of the RDTSC instruction (including RDTSCP instruction if CPUID.80000001H:EDX[27] = 1) to procedures running at privilege level 0 when set; allows RDTSC instruction (including RDTSCP instruction if CPUID.80000001H:EDX[27] = 1) to be executed at any privilege level when clear.
* DE, Debugging Extensions (bit 3 of CR4) — References to debug registers DR4 and DR5 cause an undefined opcode (#UD) exception to be generated when set; when clear, processor aliases references to registers DR4 and DR5 for compatibility with software written to run on earlier IA-32 processors.
* PSE, Page Size Extensions (bit 4 of CR4) — Enables 4-MByte pages with 32-bit paging when set; restricts 32-bit paging to pages to 4 KBytes when clear.
* PAE, Physical Address Extension (bit 5 of CR4) — When set, enables paging to produce physical addresses with more than 32 bits. When clear, restricts physical addresses to 32 bits. PAE must be set before entering IA-32e mode.
* MCE, Machine-Check Enable (bit 6 of CR4) — Enables the machine-check exception when set; disables the machine-check exception when clear.
* PGE, Page Global Enable (bit 7 of CR4) — (Introduced in the P6 family processors.) Enables the global page feature when set; disables the global page feature when clear. The global page feature allows frequently used or shared pages to be marked as global to all users (done with the global flag, bit 8, in a page-directory or page-table entry). Global pages are not flushed from the translation look aside buffer (TLB) on a task switch or a write to register CR3. When enabling the global page feature, paging must be enabled (by setting the PG flag in control register CR0) before the PGE flag is set. Reversing this sequence may affect program correctness, and processor performance will be impacted.
* PCE, Performance-Monitoring Counter Enable (bit 8 of CR4) — Enables execution of the RDPMC instruction for programs or procedures running at any protection level when set; RDPMC instruction can be executed only at protection level 0 when clear.
* VMX-Enable Bit (bit 13 of CR4) — Enables VMX operation when set.
* SMX-Enable Bit (bit 14 of CR4) — Enables SMX operation when set.
* PCID-Enable Bit (bit 17 of CR4) — Enables process-context identifiers (PCIDs) when set. Can be set only in IA-32e mode (if IA32\_EFER.LMA = 1).
* TPL, Task Priority Level (bit 3:0 of CR8) — This sets the threshold value corresponding to the highest-priority interrupt to be blocked. A value of 0 means all interrupts are enabled. This field is available in 64-bit mode. A value of 15 means all interrupts will be disabled.

**Segment Descriptor Format**

The flags and fields in a segment descriptor are as follows:

*Segment limit field*: Specifies the size of the segment. The processor puts together the two segment limit fields to form a 20-bit value. The processor interprets the segment limit in one of two ways, depending on the setting of the G (granularity) flag:

* If the granularity flag is clear, the segment size can range from 1 byte to 1 MByte, in byte increments.
* If the granularity flag is set, the segment size can range from 4 KBytes to 4 GBytes, in 4-KByte increments.

*Segment base address:* Defines the location of byte 0 of the segment within the 4-GByte linear address space. The processor puts together the three base address fields to form a single 32-bit value. Segment base addresses should be aligned to 16-byte boundaries. Although 16-byte alignment is not required, this alignment allows programs to maximize performance by aligning code and data on 16-byte boundaries.

*Type Field:* Indicates the segment or gate type and specifies the kinds of access that can be made to the segment and the direction of growth. The interpretation of this field depends on whether the descriptor type flag specifies an application (code or data) descriptor or a system descriptor. The encoding of the type field is different for code, data, and system descriptors.

*S (descriptor type) flag:* Specifies whether the segment descriptor is for a system segment (S flag is clear) or a code or data segment (S flag is set).

*DPL (descriptor privilege level) field:* Specifies the privilege level of the segment. The privilege level can range from 0 to 3, with 0 being the most privileged level. The DPL is used to control access to the segment.

*P (segment-present) flag:* Indicates whether the segment is present in memory (set) or not present (clear). If this flag is clear, the processor generates a segment-not-present exception (#NP) when a segment selector that points to the segment descriptor is loaded into a segment register.

*D/B (default operation size/default stack pointer size and/or upper bound) flag:* Performs different functions depending on whether the segment descriptor is an executable code segment, an expand-down data segment, or a stack segment. (This flag should always be set to 1 for 32-bit code and data segments and to 0 for 16-bit code and data segments.)

* Executable code segment. The flag is called the D flag and it indicates the default length for effective addresses and operands referenced by instructions in the segment. If the flag is set, 32-bit addresses and 32-bit or 8-bit operands are assumed; if it is clear, 16-bit addresses and 16-bit or 8-bit operands are assumed.
* The instruction prefix 66H can be used to select an operand size other than the default, and the prefix 67H can be used select an address size other than the default.
* Stack segment (data segment pointed to by the SS register). The flag is called the B (big) flag and it specifies the size of the stack pointer used for implicit stack operations (such as pushes, pops, and calls). If the flag is set, a 32-bit stack pointer is used, which is stored in the 32-bit ESP register; if the flag is clear, a 16-bit stack pointer is used, which is stored in the 16-bit SP register. If the stack segment is set up to be an expand-down data segment (described in the next paragraph), the B flag also specifies the upper bound of the stack segment.
* Expand-down data segment. The flag is called the B flag and it specifies the upper bound of the segment. If the flag is set, the upper bound is FFFFFFFFH (4 GBytes); if the flag is clear, the upper bound is FFFFH (64 KBytes).

*G (granularity) flag:* Determines the scaling of the segment limit field. When the granularity flag is clear, the segment limit is interpreted in byte units; when flag is set, the segment limit is interpreted in 4-KByte units.

*L (64-bit code segment) flag:* In IA-32e mode, bit 21 of the second double word of the segment descriptor indicates whether a code segment contains native 64-bit code. A value of 1 indicates instructions in this code segment are executed in 64-bit mode. A value of 0 indicates the instructions in this code segment are executed in compatibility mode. If L-bit is set, then D-bit must be cleared. When not in IA-32e mode or for non-code segments, bit 21 is reserved and should always be set to 0.

*Available and reserved bits:* Bit 20 of the second double word of the segment descriptor is available for use by system software.

When the S (descriptor type) flag in a segment descriptor is clear, the descriptor type is a system descriptor. The processor recognizes the following types of system descriptors:

* Local descriptor-table (LDT) segment descriptor.
* Task-state segment (TSS) descriptor.
* Call-gate descriptor.
* Interrupt-gate descriptor.
* Trap-gate descriptor.
* Task-gate descriptor.

These descriptor types fall into two categories: system-segment descriptors and gate descriptors. System-segment descriptors point to system segments (LDT and TSS segments). Gate descriptors are in themselves “gates,” which hold pointers to procedure entry points in code segments (call, interrupt, and trap gates) or which hold segment selectors for TSS’s (task gates).

**VT**

**VMCS**

* A 4K-aligned memory buffer for saving/restoring CPU state
* Comprised of
  + VMCS Region
  + VMCS data
    - Guest state : to save/load guest state on VMexit/VMentry
    - Host state: to load host state on VMexit
    - VMX execution control fields: configure VMentry/exit conditions
    - VMentry control fields: to control specific saving/loading of guest on VMentry
    - VMexit control fields: to control specific loading of host state on VMexit
    - VMexit information fields: contain information about the cause and the nature of the Vmexit
* 3 new registers added on each thread for VMCS management
  + Parent, working-VMCS, and controlling-VMCS pointers
* There could be more than 1 valid VMCS at a time, but only one active (working-vmcs pointer)
* Loading a VMCS into working-VMCS pointer by executing *vmptrld* instruction

**Transitions**

**Page Walk, Mode L (4K), with EPT (4K)**

**VMX Basics**

**Secured Environment Model (TXT)**

* Monitor provides domain separation, VM entry / exit policy enforcement, and inter-VM communications channels.
* Trusted kernel provides intra-VM services; may be designed to interact with a specific main OS.

**SMM Peer Monitor Overview**

* VMM needs to take these actions:
  + Initially enter VMX mode (using ‘**vmxon**’)
  + Clear our guest’s VMCS (using ‘**vmclear**’)
  + Load the guest-pointer (using ‘**vmptrld**’)
  + Write VMCS parameters (using ‘**vmwrite**’)
  + Launch the guest-task (using ‘**vmlaunch**’)
  + Read guest-exit information (using ‘**vmread**’)
  + Maybe reenter the guest (using ‘**vmresume**’)
  + Eventually leave VMX mode (using ‘**vmxoff**’)
* We need these VMM data-structures:
  + 4-level page-tables (required for 64-bit code)
  + Task-State Segment (no permission-bitmap)
  + Global Descriptor Table (code, data, TSS)
  + Interrupt Descriptor Table (for GP faults)
  + One stack-region (for VMM’s ‘ring0’ code)
  + Virtual Machine Control Structures (two --- host and guest)
  + Collection of variables (VMCS parameters)