cs5965 Advanced OS Implementation

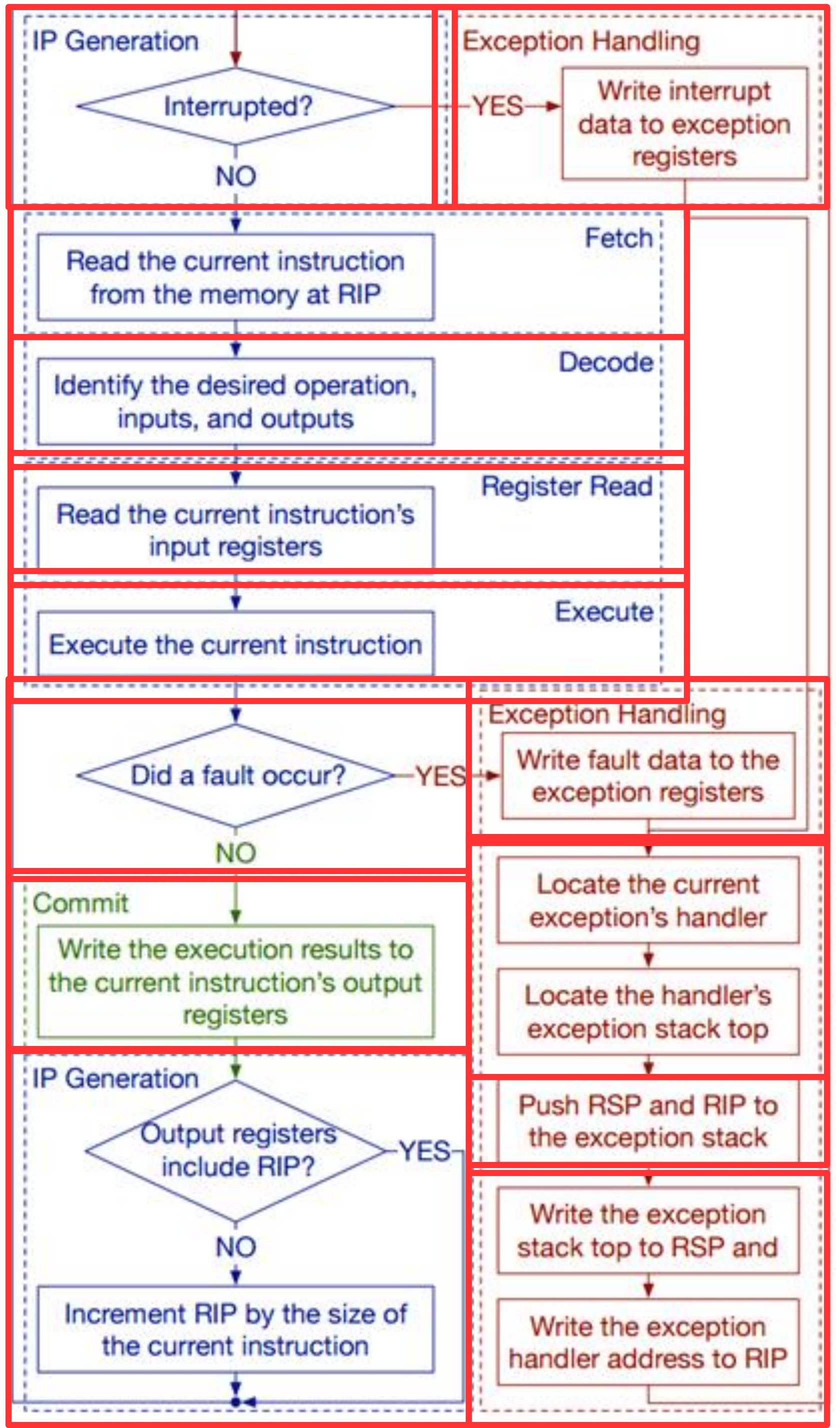
Lecture 04 – Interrupts and Exceptions

Anton Burtsev

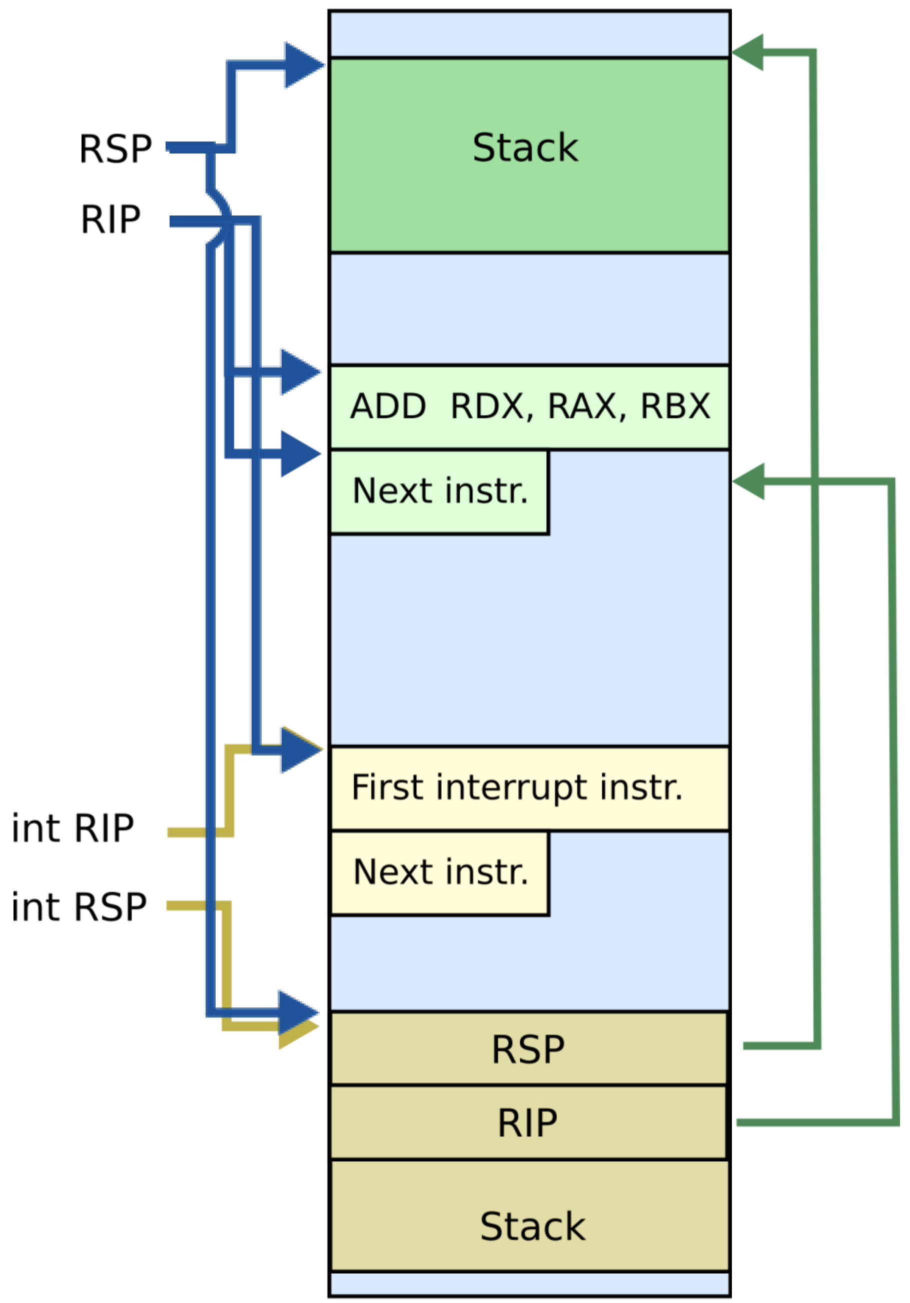
Why do we need interrupts?

- Two main use cases:
  - [Synchronous] Something bad happened and OS needs to fix it
    - Program tries to access an unmapped page
  - [Asynchronous] Notifications from external devices
    - Network packet arrived (OS will copy the packet from temporary buffer in memory (to avoid overflowing) and may switch to a process waiting on that packet)
    - Timer interrupt (OS may switch to another process)
- A third, special, use-case
  - [It's also synchronous] For many years an interrupt, e.g., int 0x80 instruction, was used as a mechanism to transfer control flow from user-level to kernel in a secure manner
  - This was used to implement system calls
  - Now, a faster mechanism is available (sysenter)

- How do we handle an interrupt?



IP Generation
Interrupted?
YES
NO
Exception Handling
Write interrupt data to exception registers
Fetch
Read the current instruction from the memory at RIP
Decode
Identify the desired operation, inputs, and outputs
Register Read
Read the current instruction's input registers
Execute
Execute the current instruction
Did a fault occur?
YES
NO
Exception Handling
Write fault data to the exception registers
Commit
Write the execution results to the current instruction's output registers
Locate the current exception's handler
Locate the handler's exception stack top
Push RSP and RIP to the exception stack
IP Generation
Output registers include RIP?
YES
NO
Increment RIP by the size of the current instruction
Write the exception stack top to RSP and
Write the exception handler address to RIP



RSP
RIP
Stack
ADD RDX, RAX, RBX
Next instr.
int RIP
int RSP
First interrupt instr.
Next instr.
RSP
RIP
Stack

Handling interrupts and exceptions

- In both synchronous and asynchronous cases the CPU follows the **same procedure**
  - Stop execution of the current program
  - Start execution of a handler
  - Processor accesses the handler through an entry in the Interrupt Descriptor Table (IDT)
    - Each interrupt is defined by a number
  - E.g., 14 is page fault, 3 debug
  - This number is an index into the interrupt table (IDT)

There might be two cases

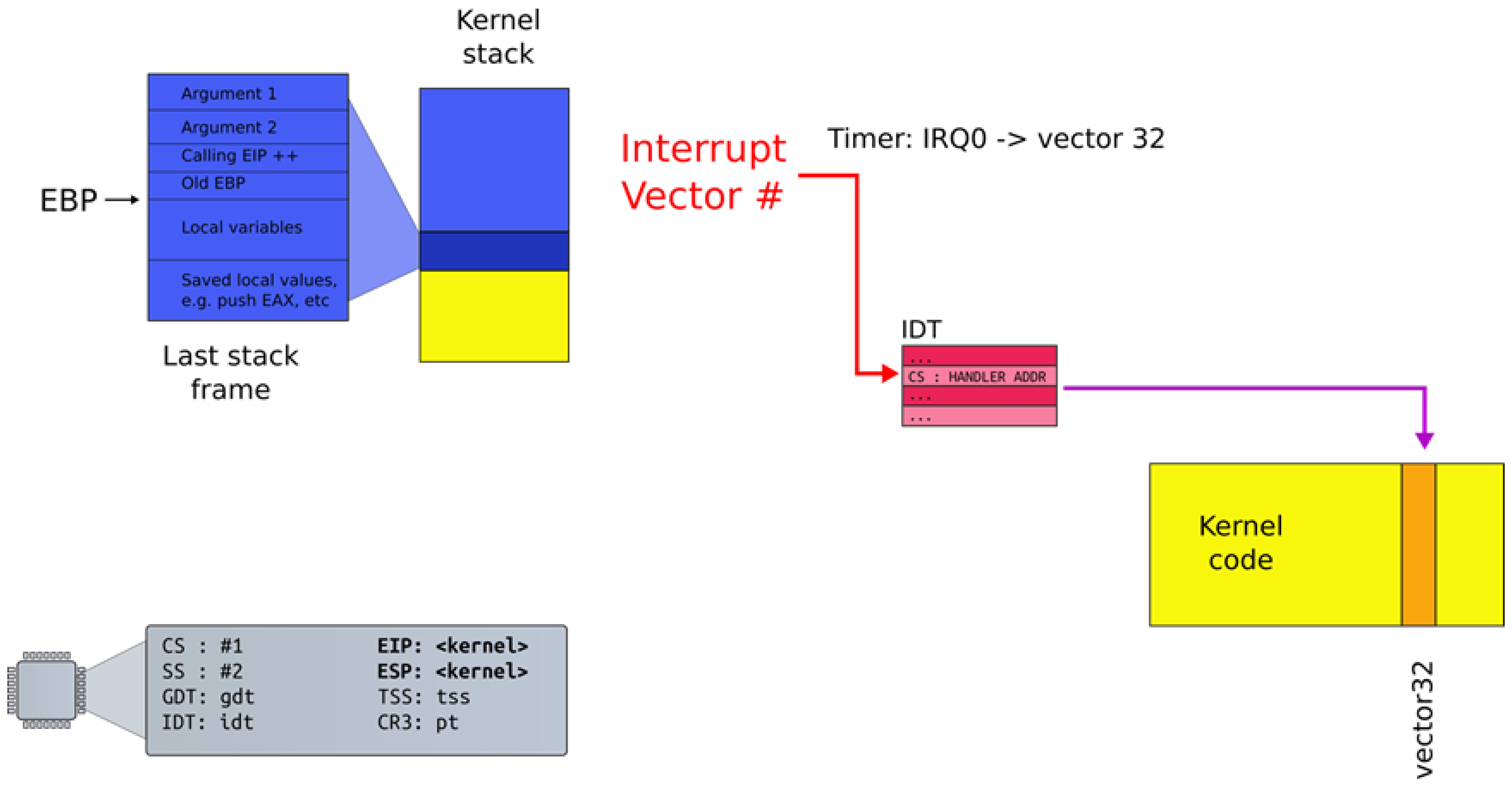
- Interrupt requires **no change** of privilege level
    - i.e., the CPU runs kernel code (privilege level 0) when a timer interrupt arrives, or kernel tries to access an unmapped page
- Interrupt **changes** privilege level
    - i.e., the CPU runs **user** code (privilege level 3) when a timer interrupt arrives, or
    - user code tries to access an unmapped page

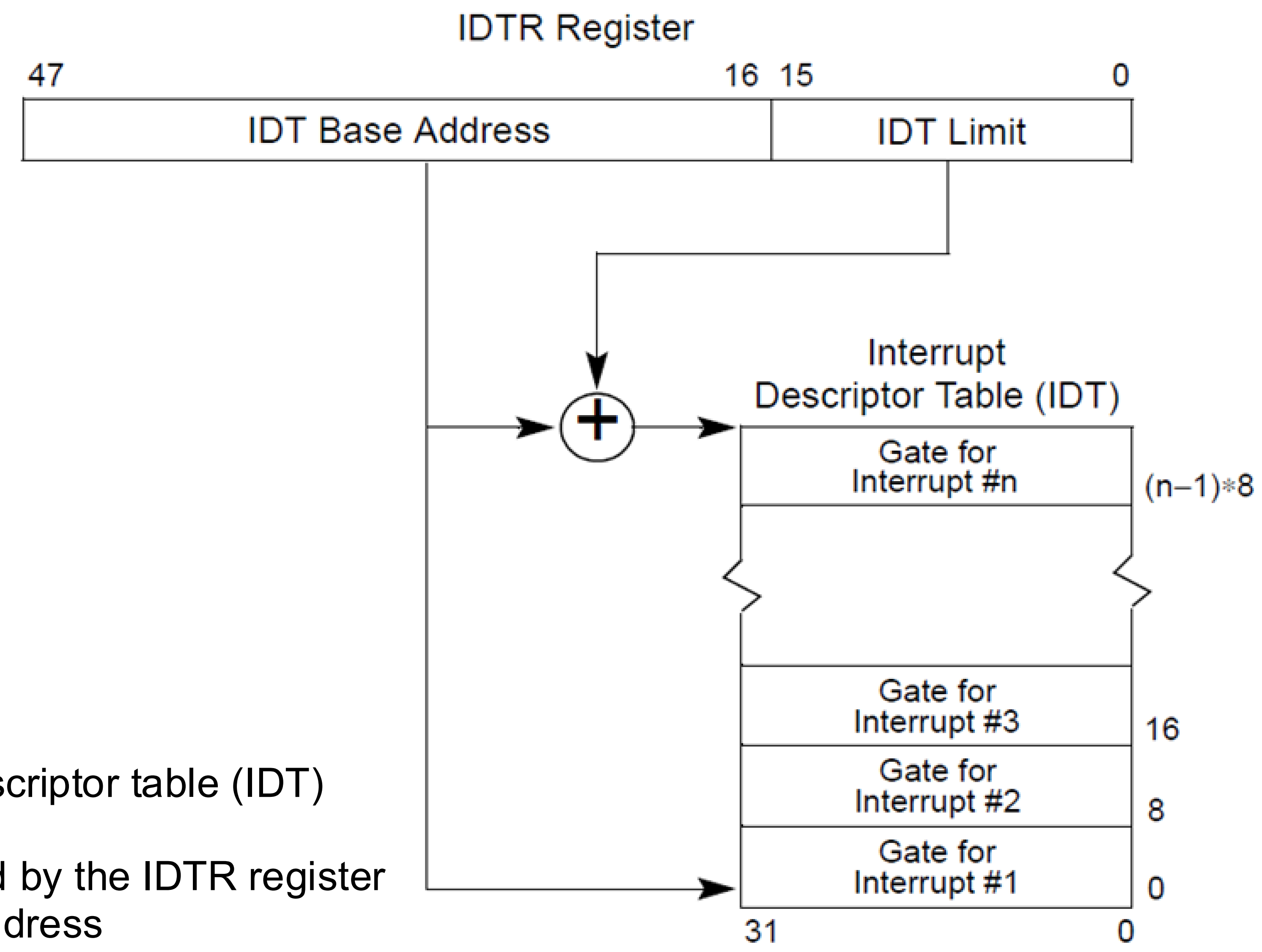
I will first explain how interrupts work and then talk about privilege levels

It’s easier this way

Case #1: Interrupt path no change in privilege level

- e.g., we're already running in the kernel

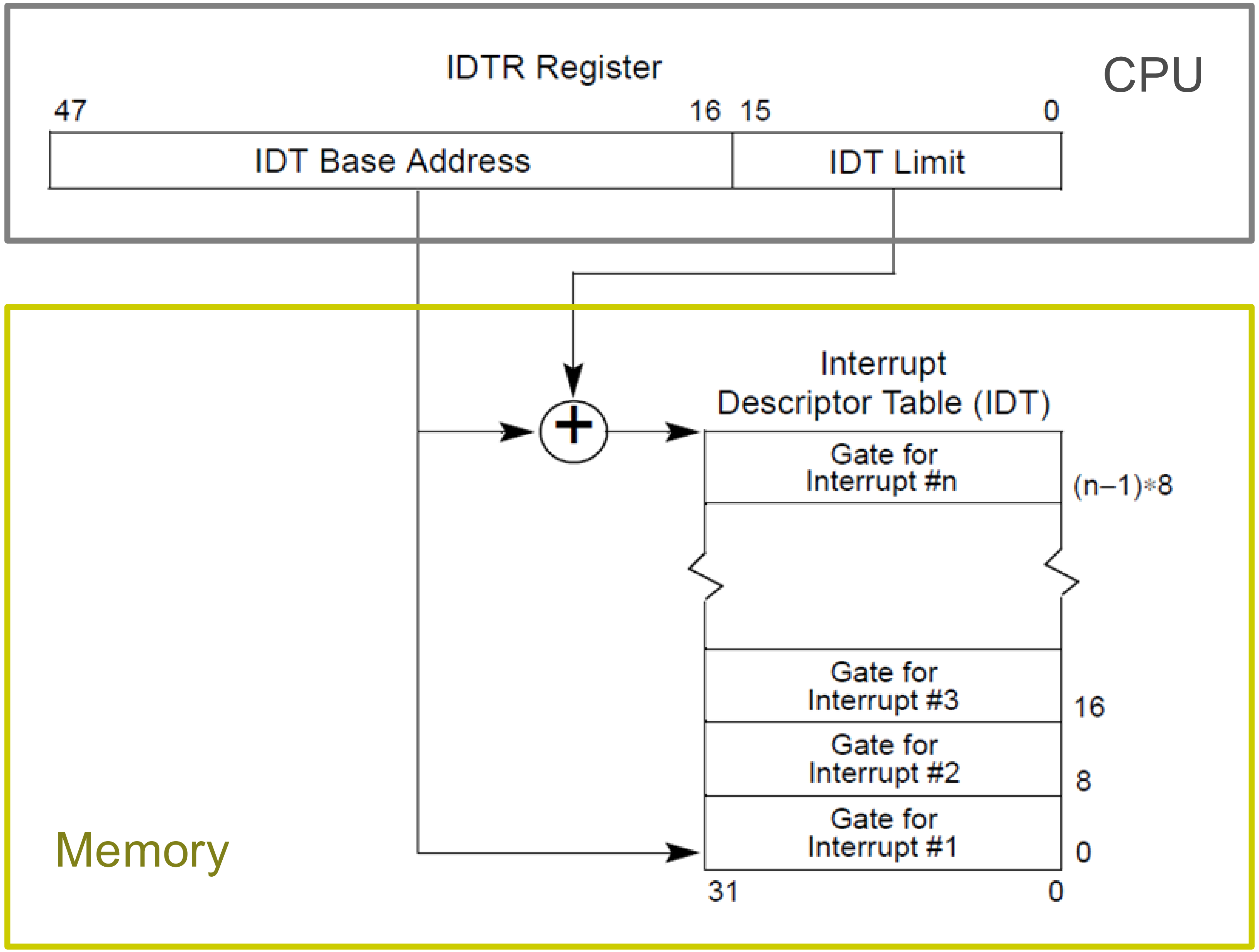




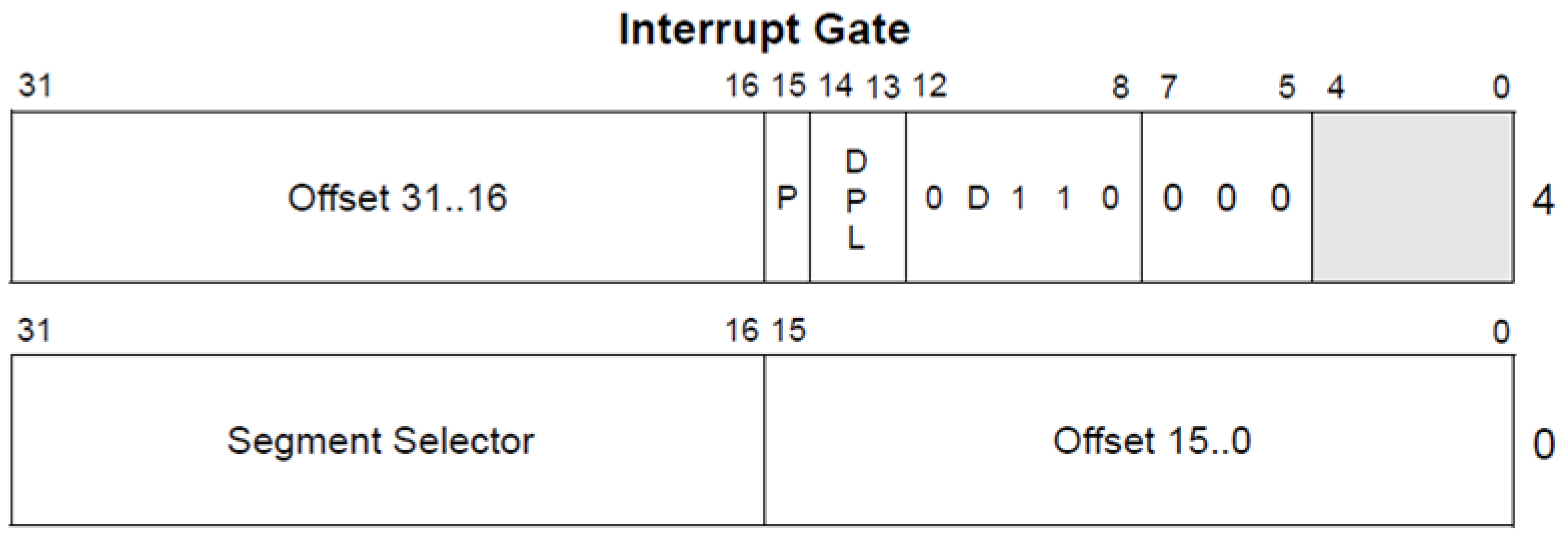
Interrupt descriptor table (IDT)

- Is pointed by the IDTR register
- Virtual address

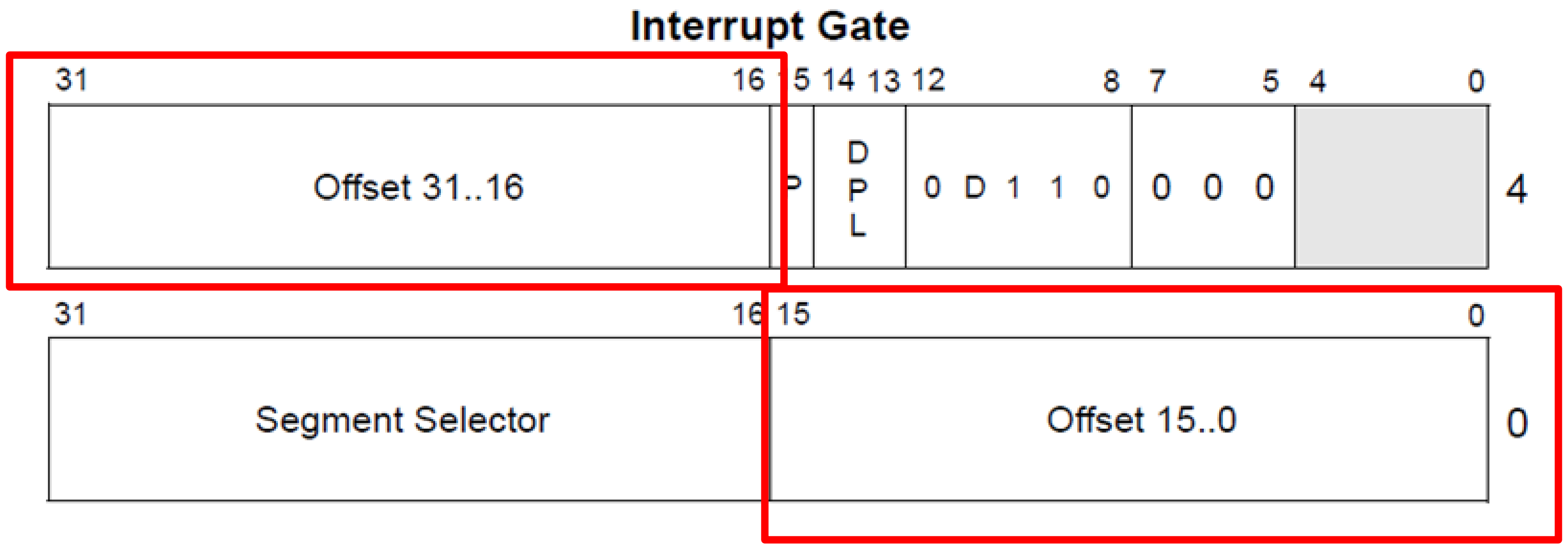
- OS configures the value and loads it into the register (normally during boot)


IDTR Register
CPU
47
16
15
0
IDT Base Address
IDT Limit
+
Interrupt Descriptor Table (IDT)
Gate for Interrupt #n
(n–1)*8
Gate for Interrupt #3
16
Gate for Interrupt #2
8
Gate for Interrupt #1
0
31
0
Memory

Interrupt descriptor



Interrupt descriptor

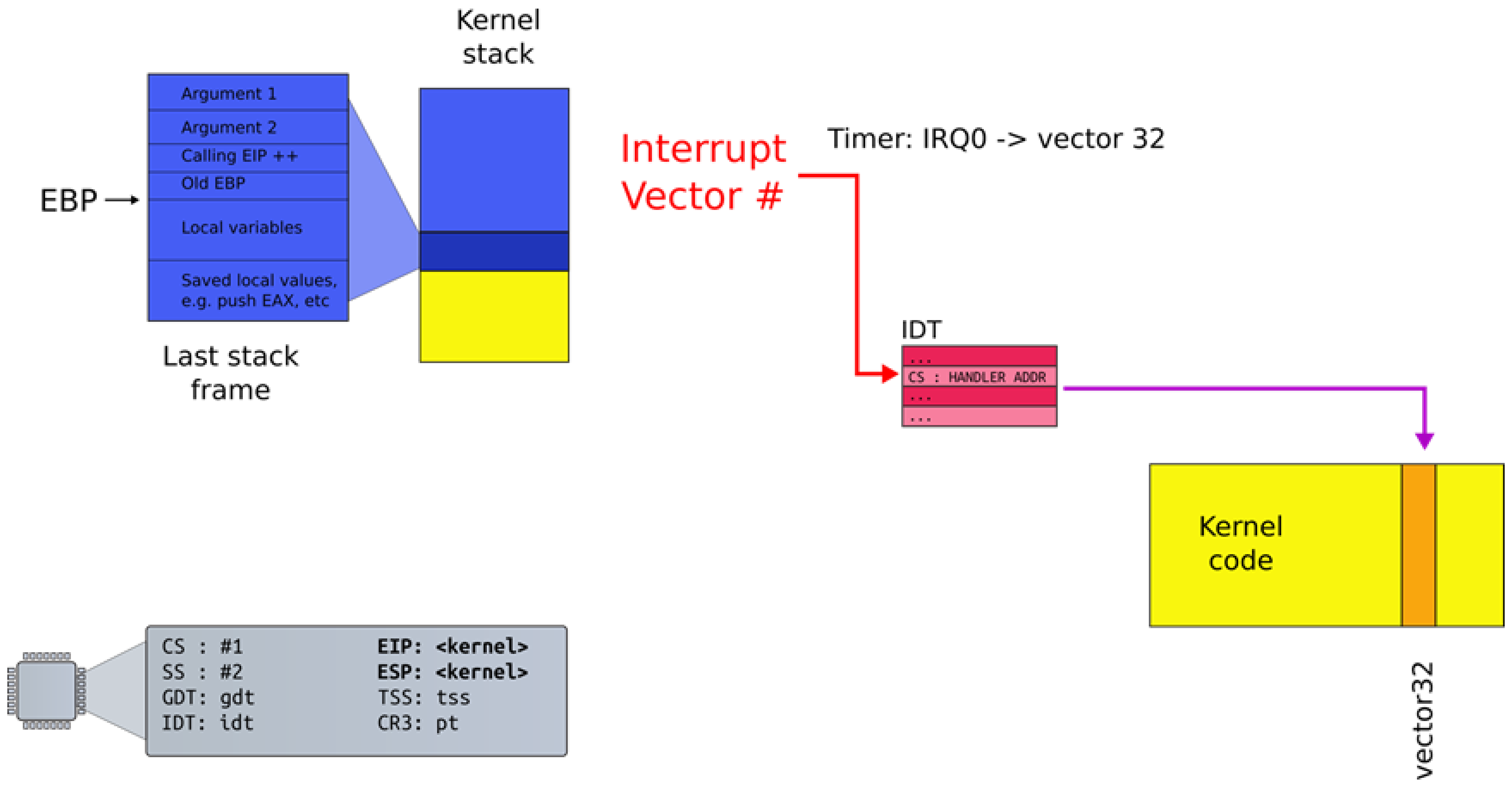


- We will walk through these fields gradually
- For now, we care about vector offset
    - Pointer to the interrupt handler

Interrupt handlers

- Just plain old code in the kernel
- The IDT stores a pointer to the right handler routine

- Interrupt path

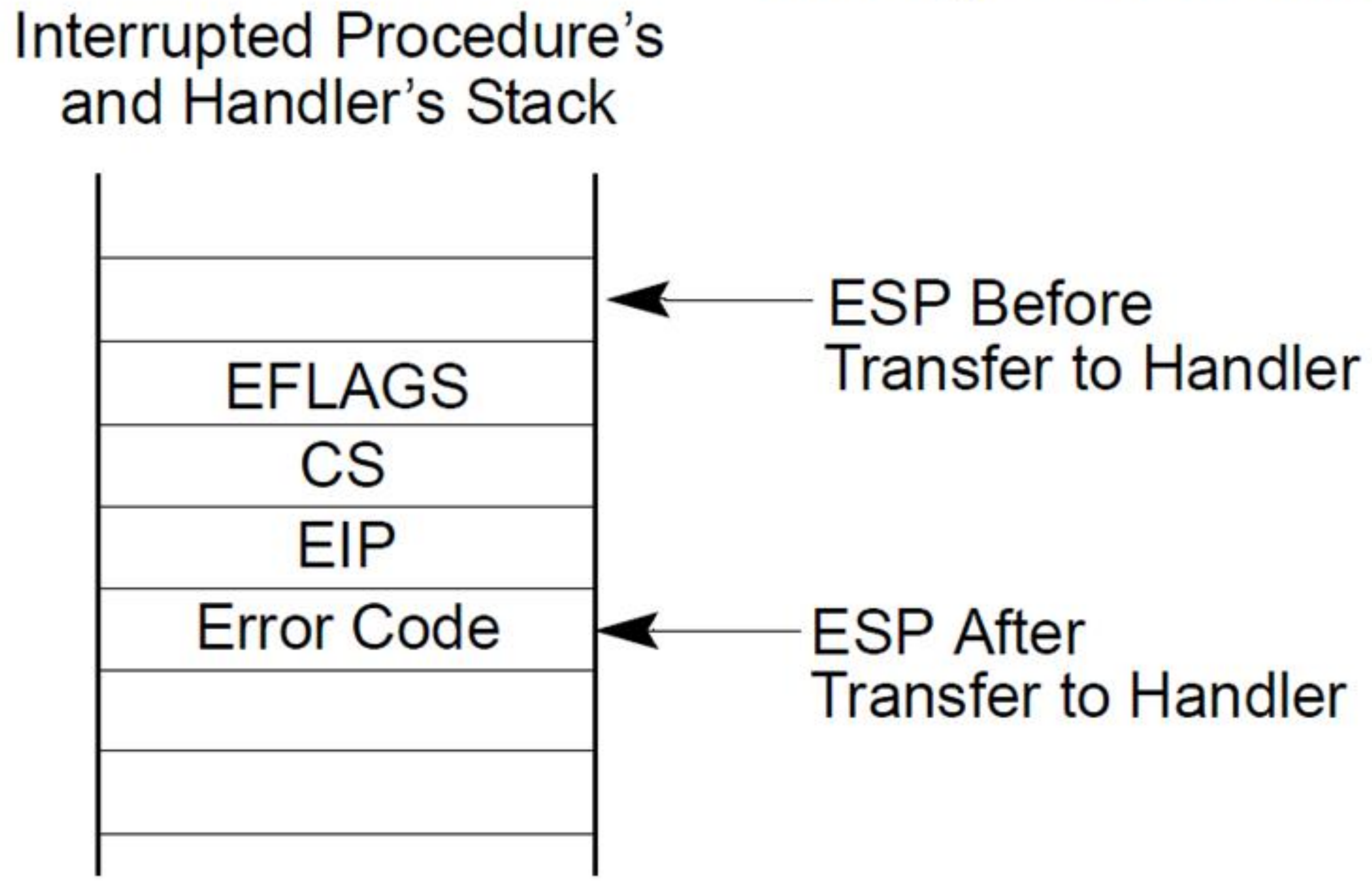


```
CS : #1          EIP: <kernel>
SS : #2          ESP: <kernel>
GDT: gdt         TSS: tss
IDT: idt         CR3: pt
```

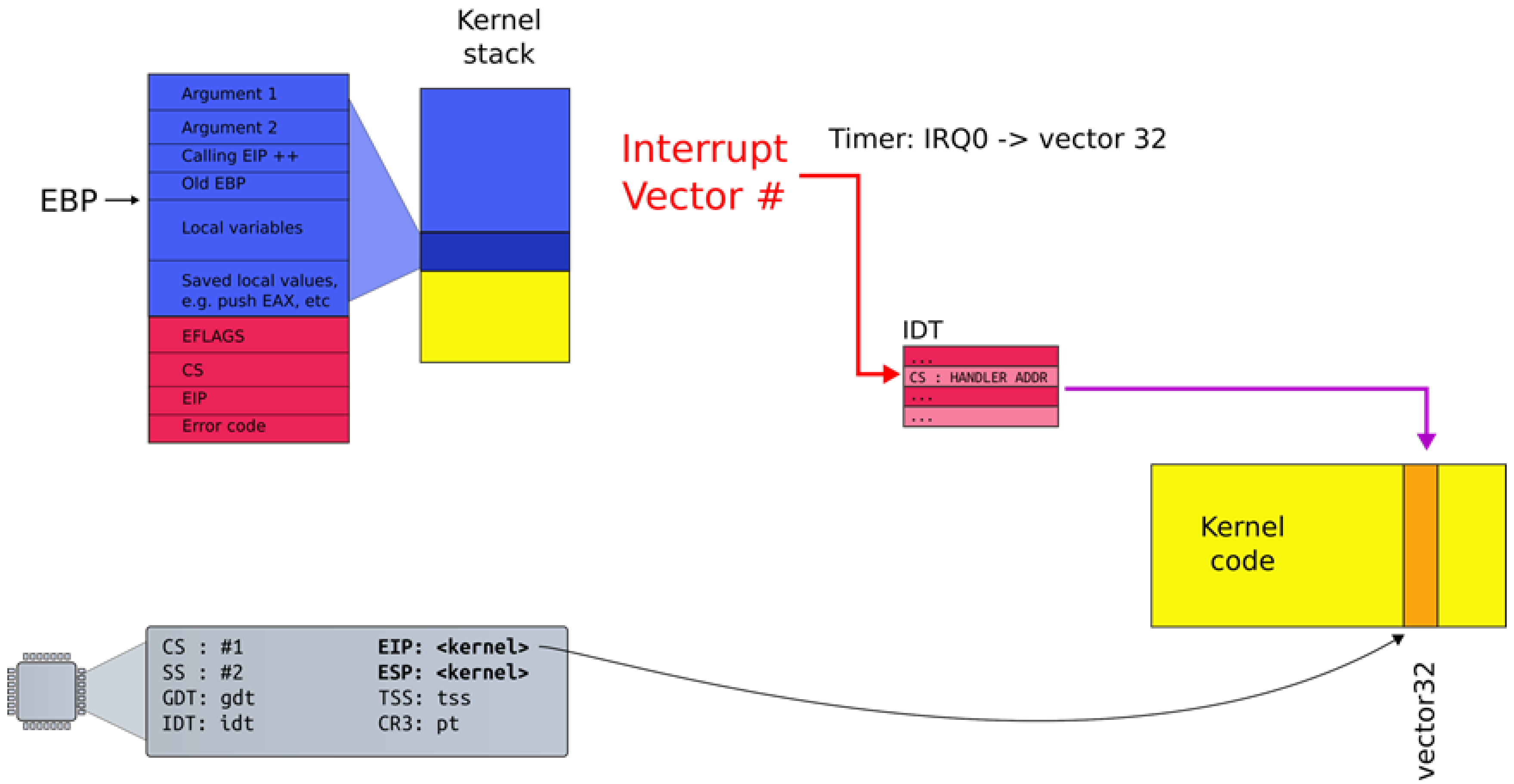
Processing of interrupt (same PL)

1. Push the current contents of the EFLAGS, CS, and EIP registers (in that order) on the stack
2. Push an error code (if appropriate) on the stack
3. Load the segment selector for the new code segment and the new instruction pointer (from the interrupt gate or trap gate) into the CS and EIP registers
4. If the call is through **an interrupt gate**, clear the IF flag in the EFLAGS register (**disable further interrupts**)
5. Begin execution of the handler

Stack Usage with No Privilege-Level Change


Interrupted Procedure's and Handler's Stack
EFLAGS
CS
EIP
Error Code
ESP Before Transfer to Handler
ESP After Transfer to Handler

- Interrupt path



Return from an interrupt

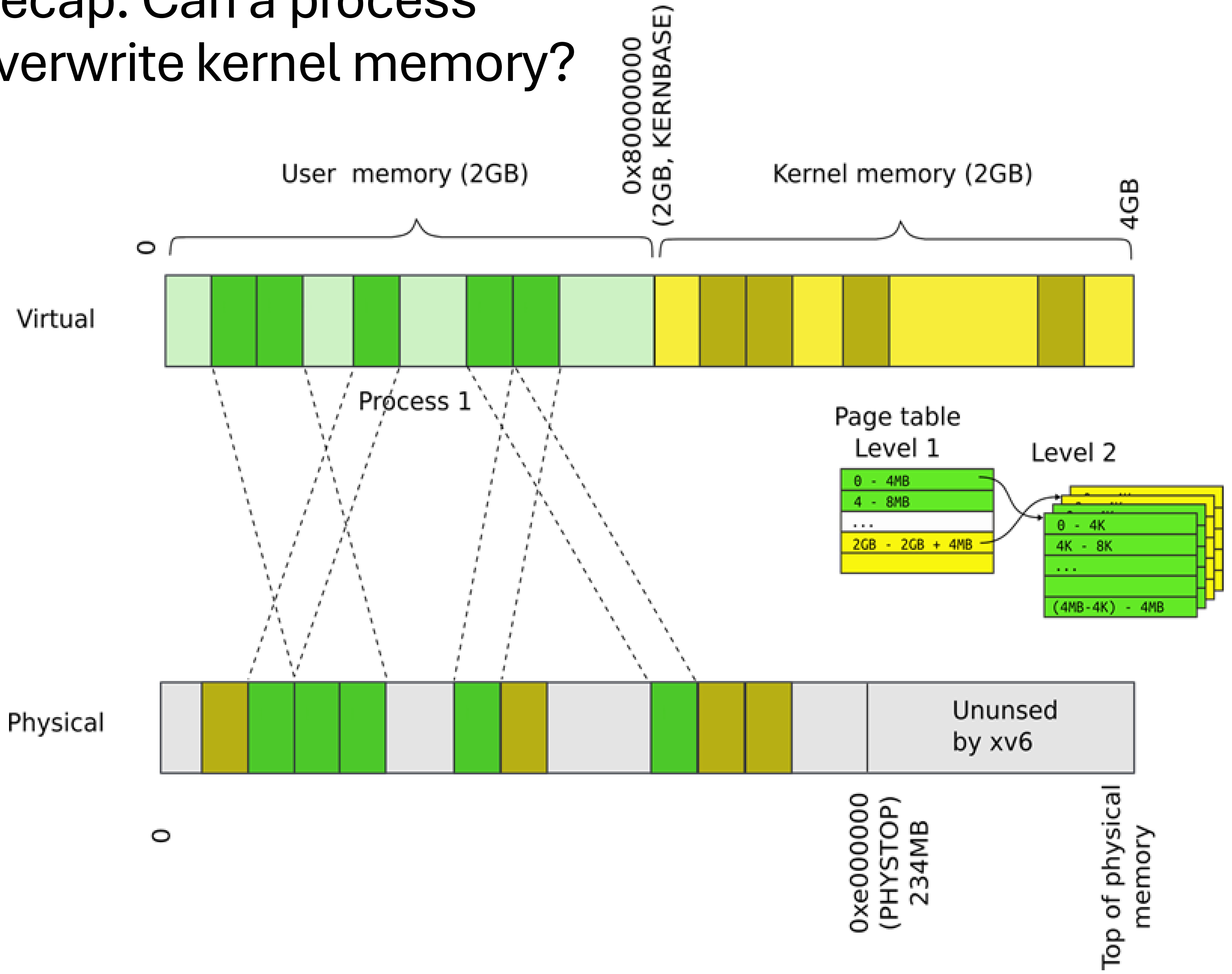
1. Starts with IRET
2. Restore the CS and EIP registers to their values prior to the interrupt or exception
3. Restore EFLAGS
4. Restore SS and ESP to their values prior to interrupt
   - This results in a stack switch
5. Resume execution of interrupted procedure

Poll: PollEv.com/antonburtsev

- Which registers are saved on interrupt transition?

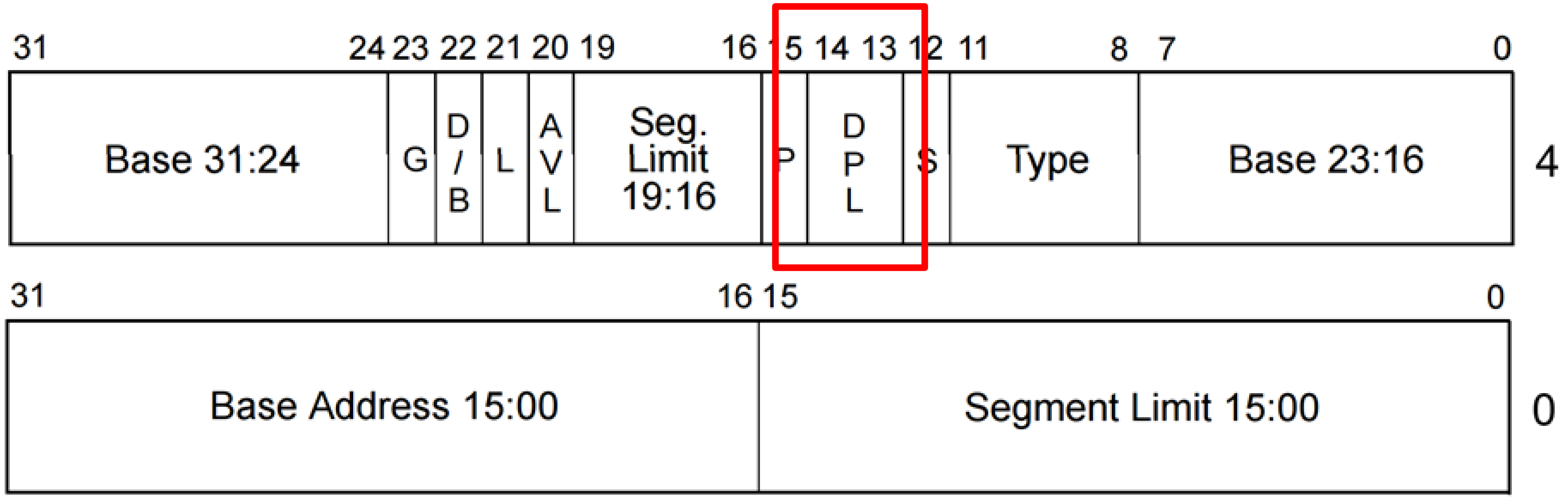
- Detour:
- What are those privilege levels?

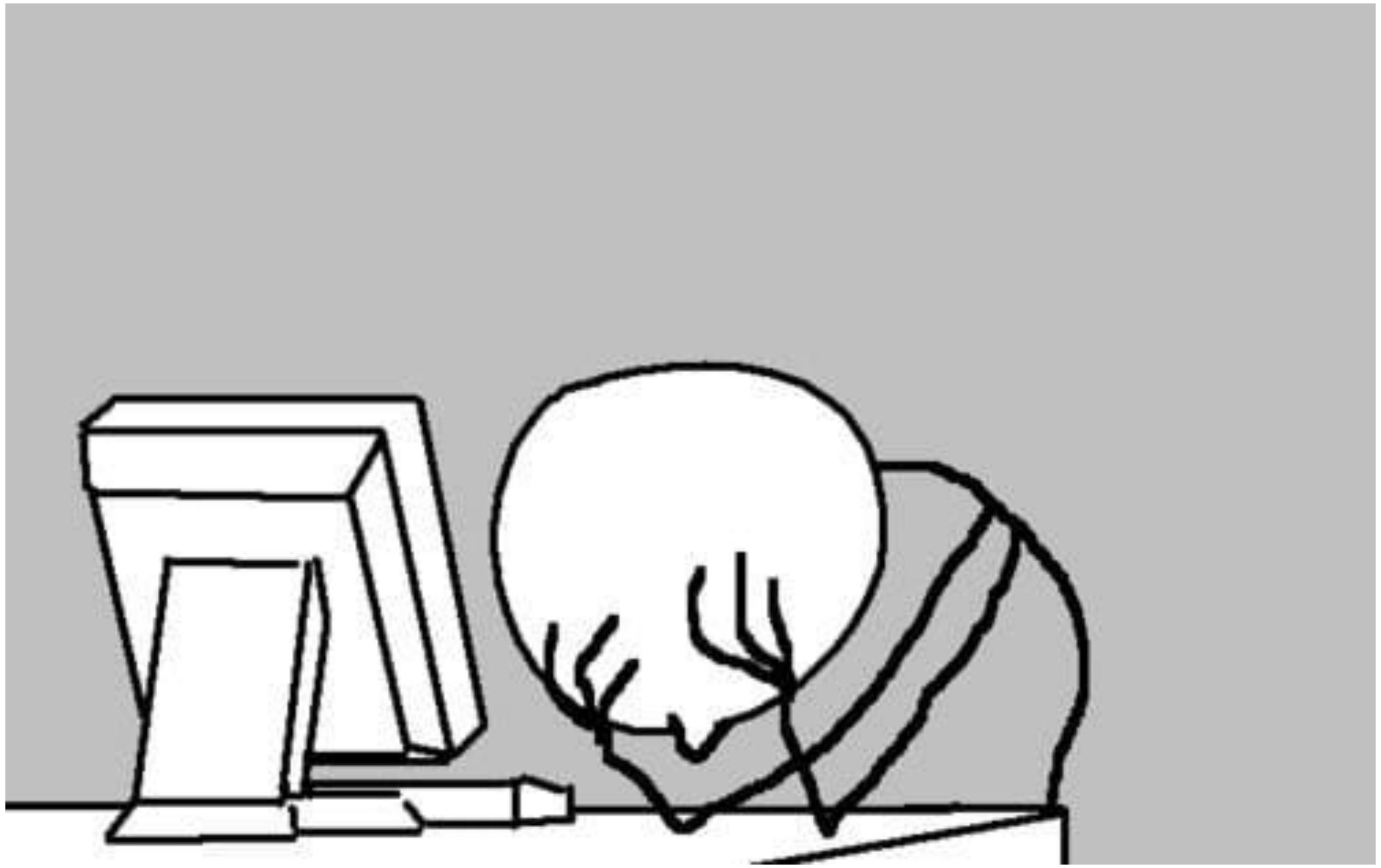
Recap: Can a process overwrite kernel memory?

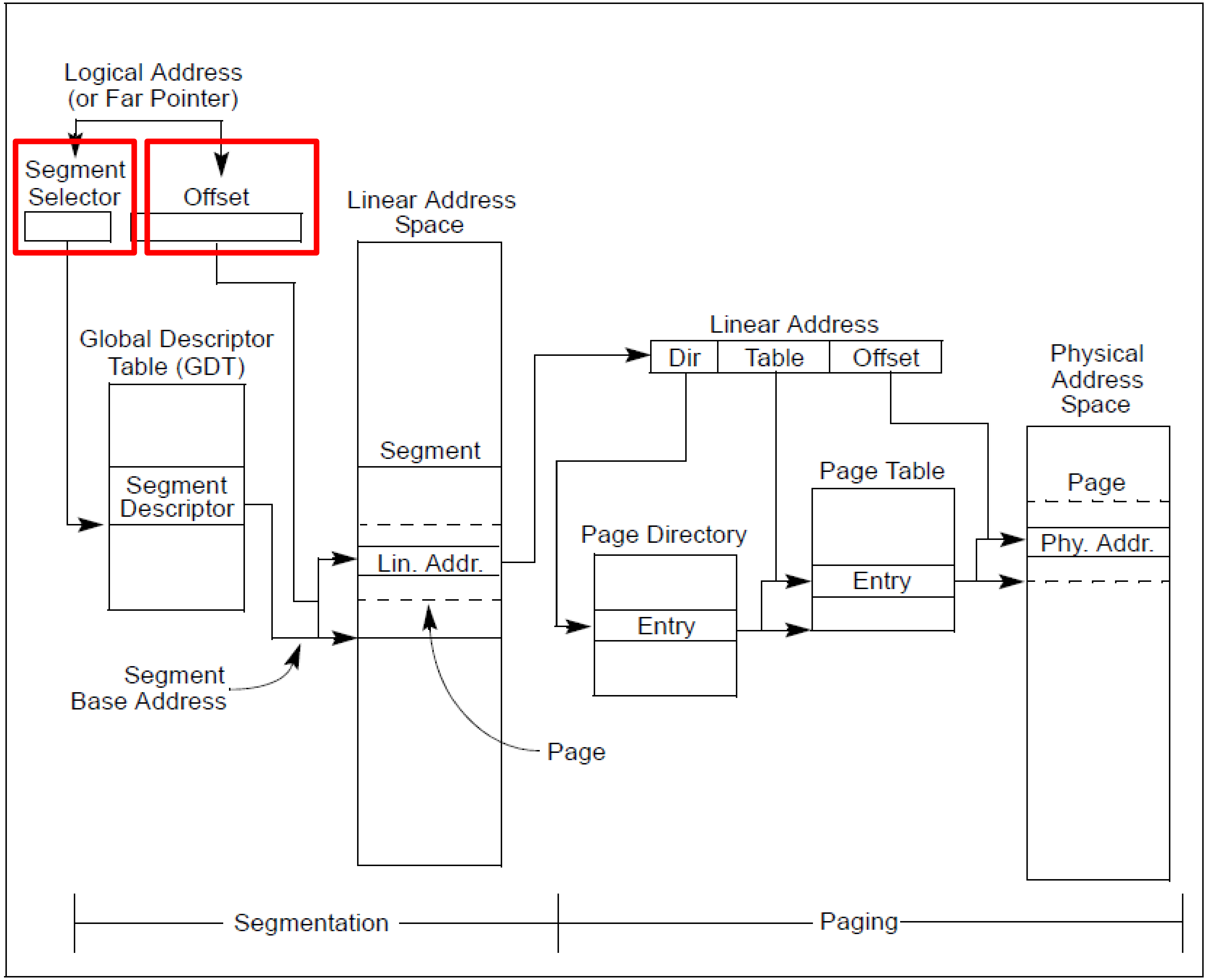


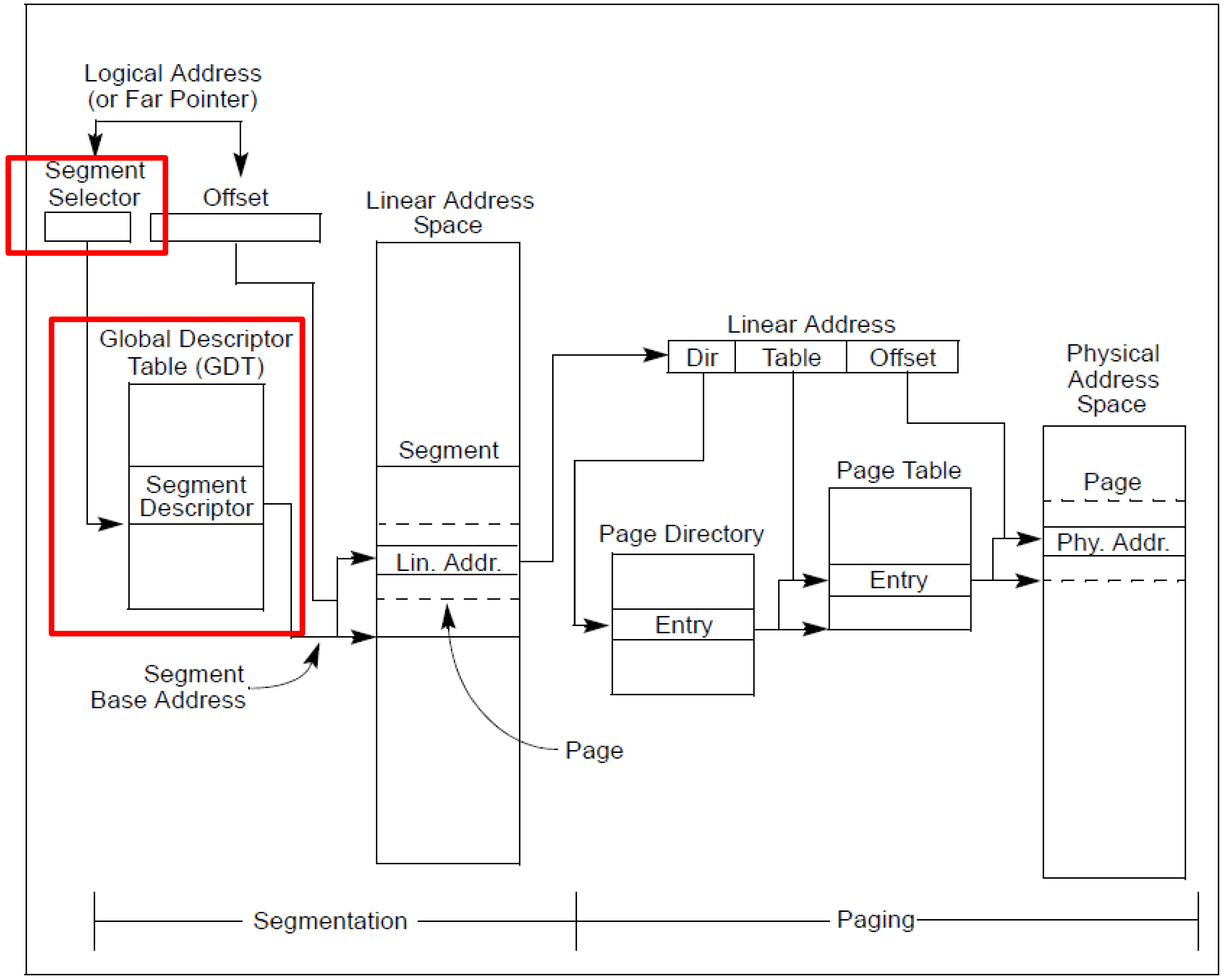
Privilege levels

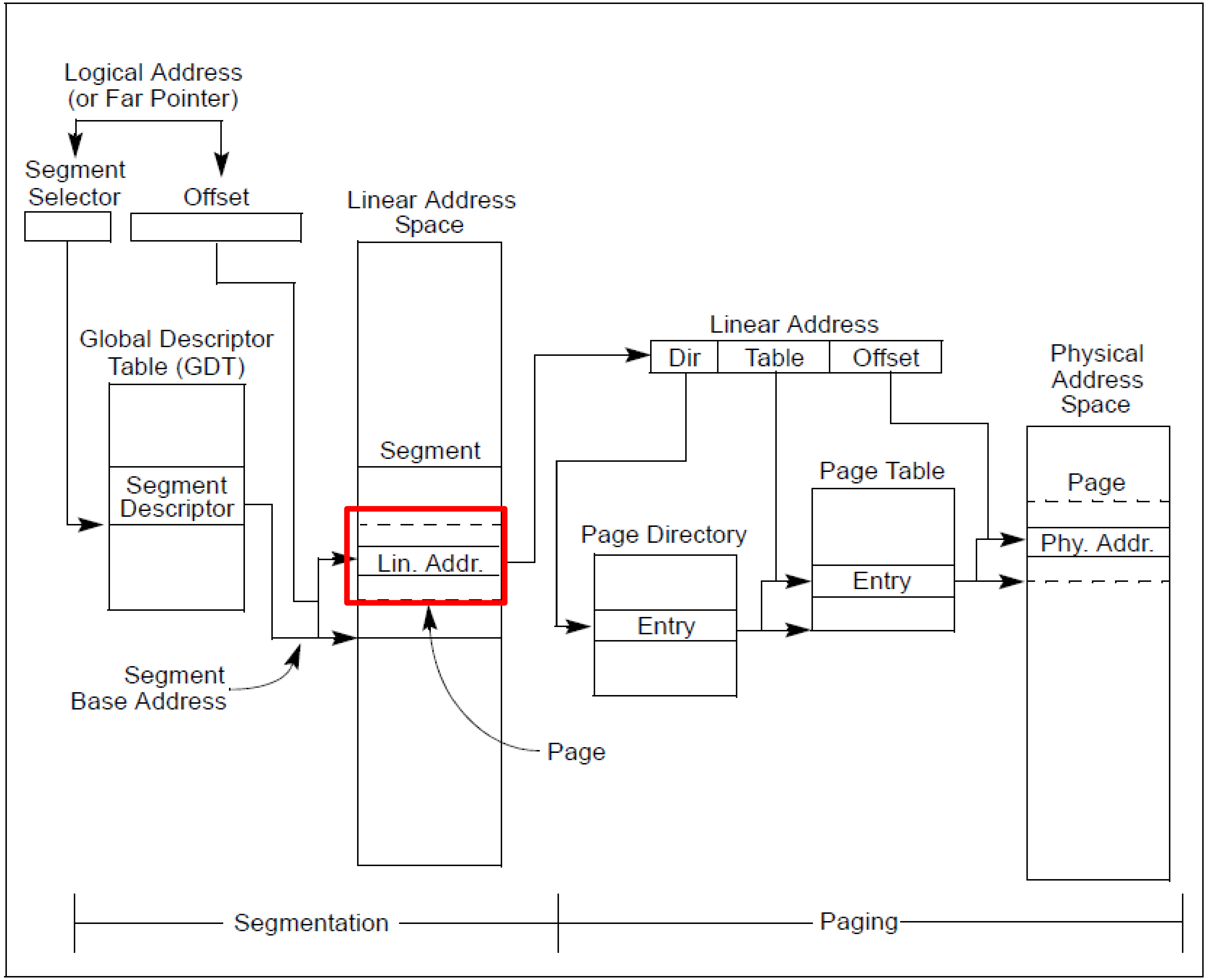
- Each segment has a privilege level
- DPL (descriptor privilege level)
- 4 privilege levels ranging 0-3





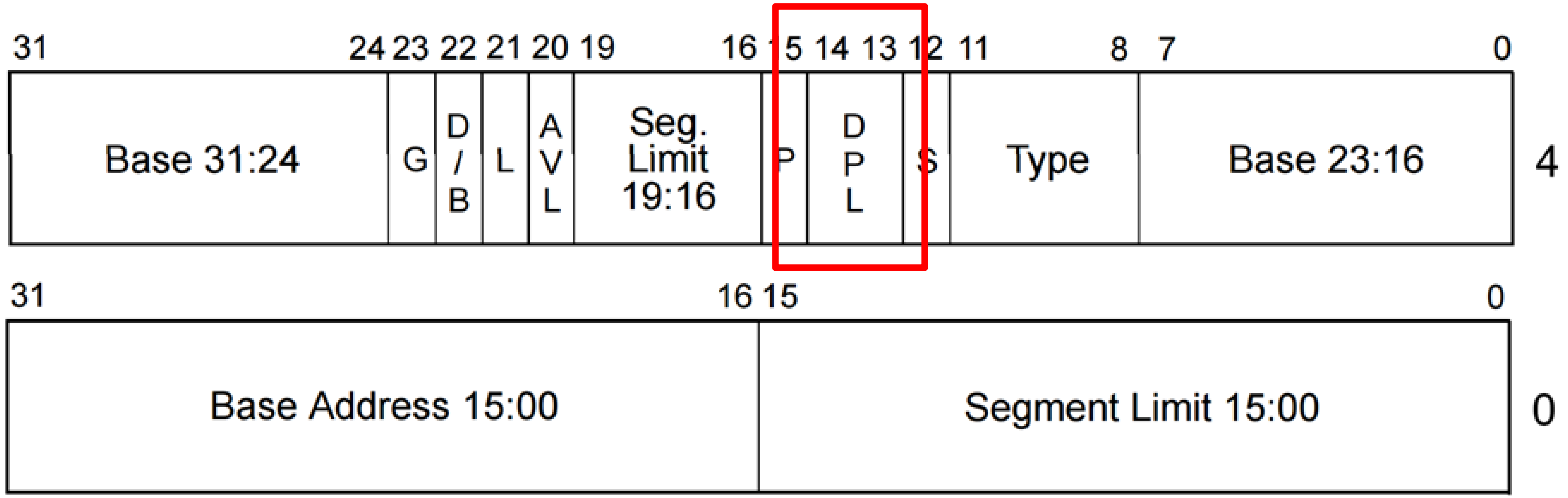

Logical Address
(or Far Pointer)
Segment Selector
Offset
Linear Address Space
Global Descriptor Table (GDT)
Segment Descriptor
Segment
Lin. Addr.
Segment Base Address
Page
Linear Address
Dir
Table
Offset
Page Directory
Entry
Page Table
Entry
Physical Address Space
Page
Phy. Addr.
Segmentation
Paging


Logical Address (or Far Pointer)
Segment Selector
Offset
Linear Address Space
Global Descriptor Table (GDT)
Segment Descriptor
Segment
Lin. Addr.
Segment Base Address
Page
Linear Address
Dir
Table
Offset
Page Directory
Entry
Page Table
Entry
Physical Address Space
Page
Phy. Addr.
Segmentation
Paging


Logical Address
(or Far Pointer)
Segment Selector
Offset
Linear Address Space
Global Descriptor Table (GDT)
Segment Descriptor
Segment
Lin. Addr.
Segment Base Address
Page
Linear Address
Dir
Table
Offset
Page Directory
Entry
Page Table
Entry
Physical Address Space
Page
Phy. Addr.
Segmentation
Paging

Privilege levels

- Each segment has a privilege level
- DPL (descriptor privilege level)
- 4 privilege levels ranging 0-3



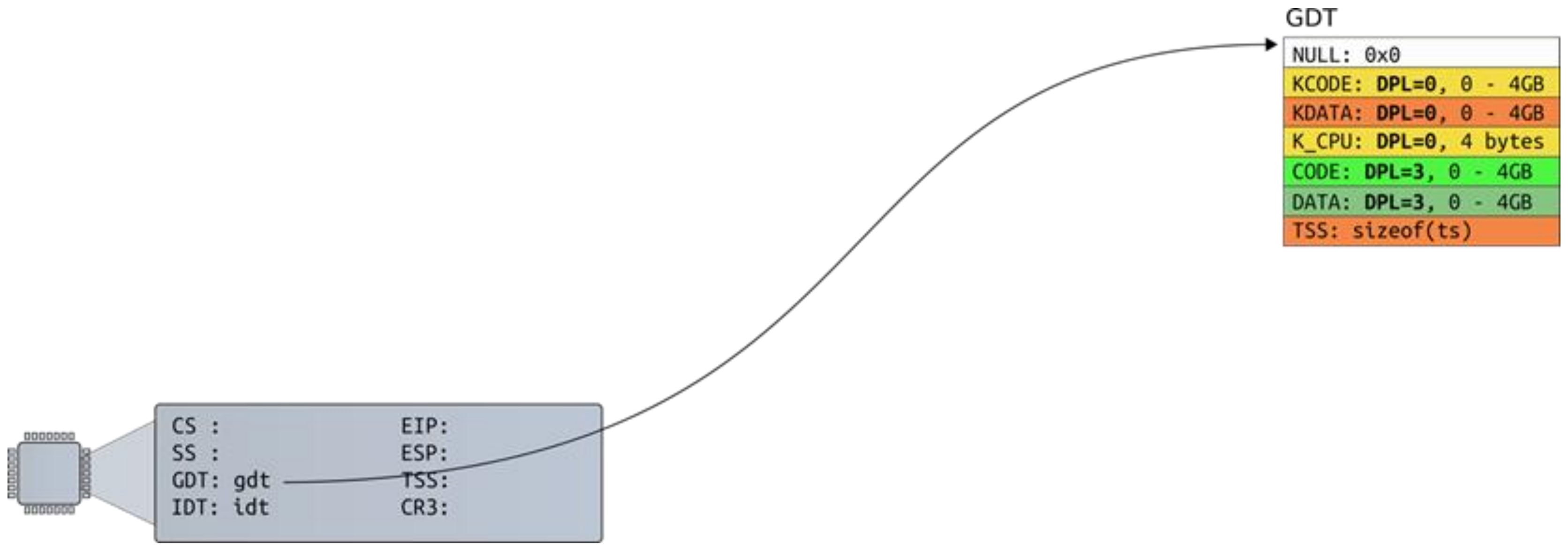
Privilege levels

- Currently running code also has a privilege level
- “Current privilege level” (CPL): 0-3
- It is saved in the CS register
  - It was loaded there when the descriptor for the currently running code was loaded into CS

Privilege level transitions

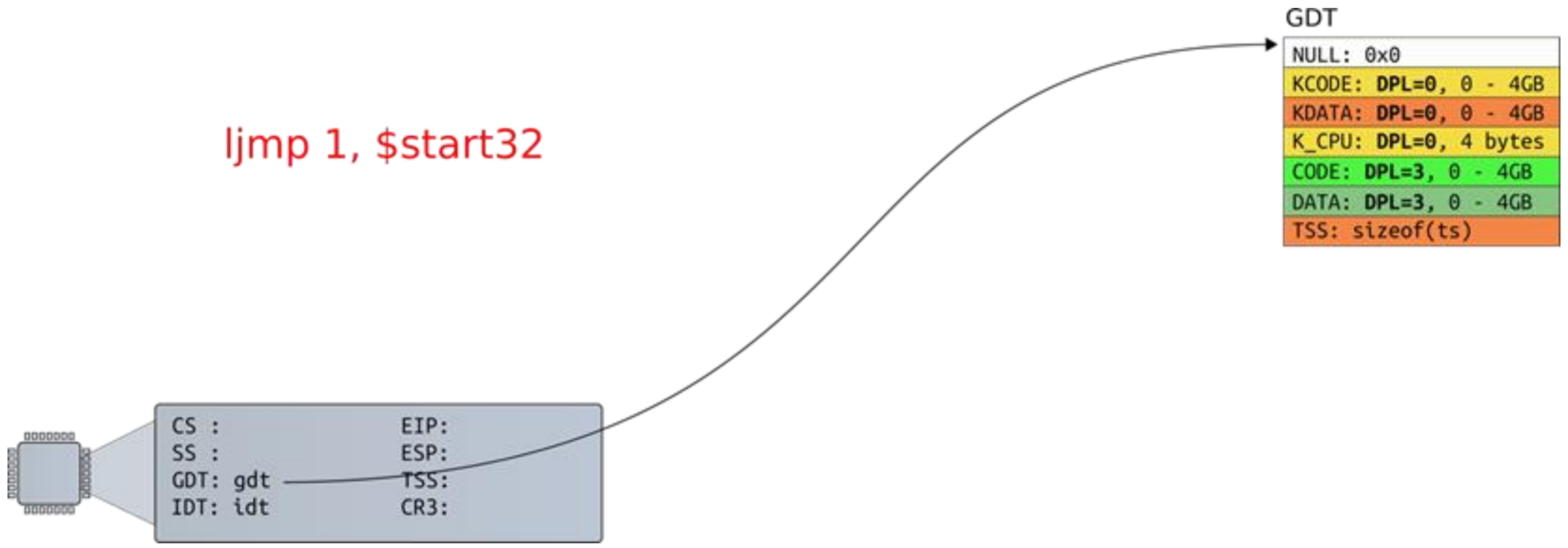
- CPL can access only less privileged segments
  - E.g., 0 can access 0, 1, 2, 3
  - 1 can access 1, 2, 3
  - 3 can access 3
- Some instructions are “privileged”
  - Can only be invoked at CPL = 0
  - Examples:
    - Load GDT
    - MOV <control register>
      - E.g. reload a page table by changing CR3

Xv6 example: started boot (no CPL yet)



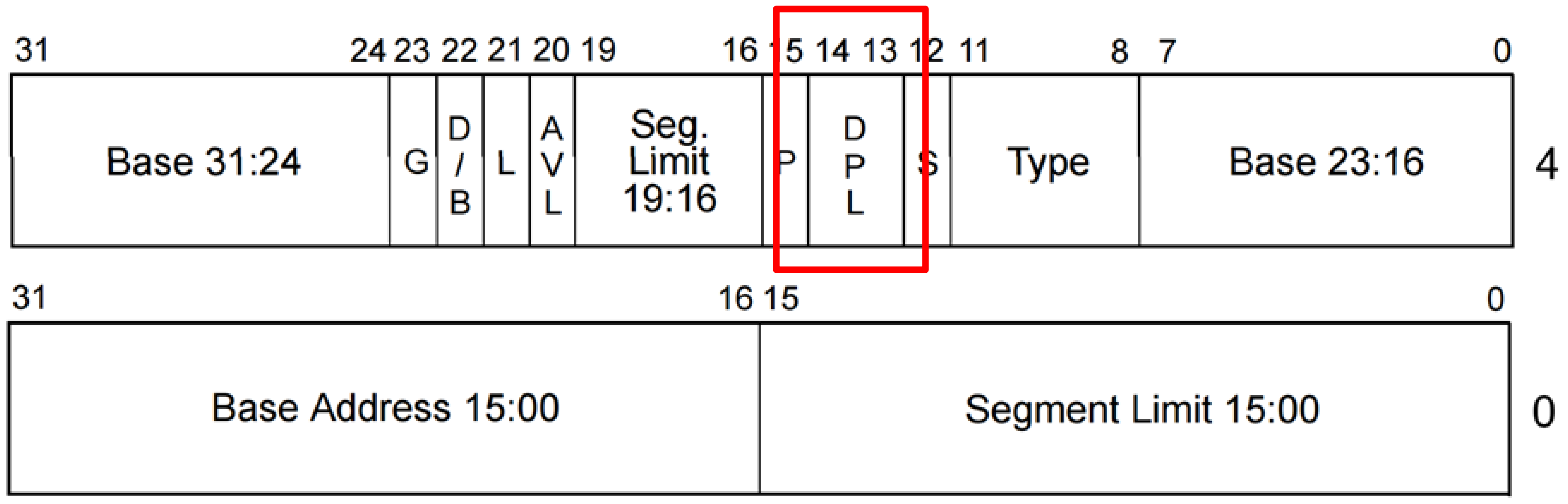
Xv6 example: prepare to load GDT entry #1

ljmp 1, $start32



Privilege levels

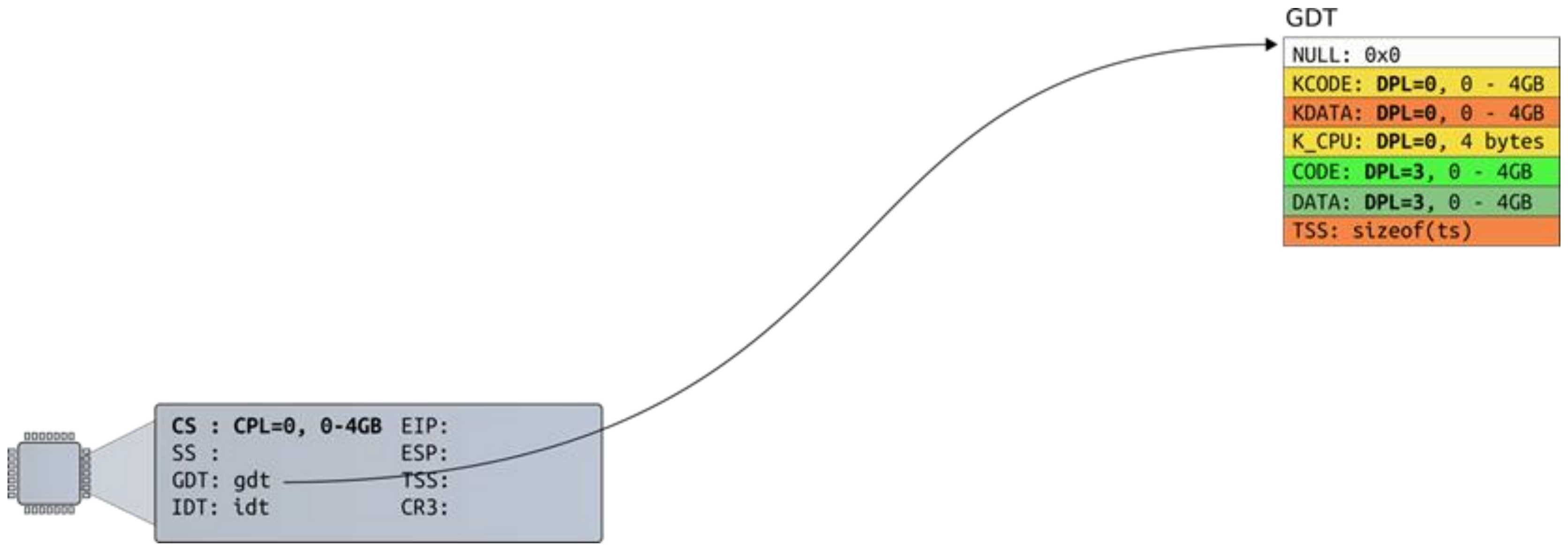
- Each segment has a privilege level
- DPL (descriptor privilege level)
- 4 privilege levels ranging 0-3



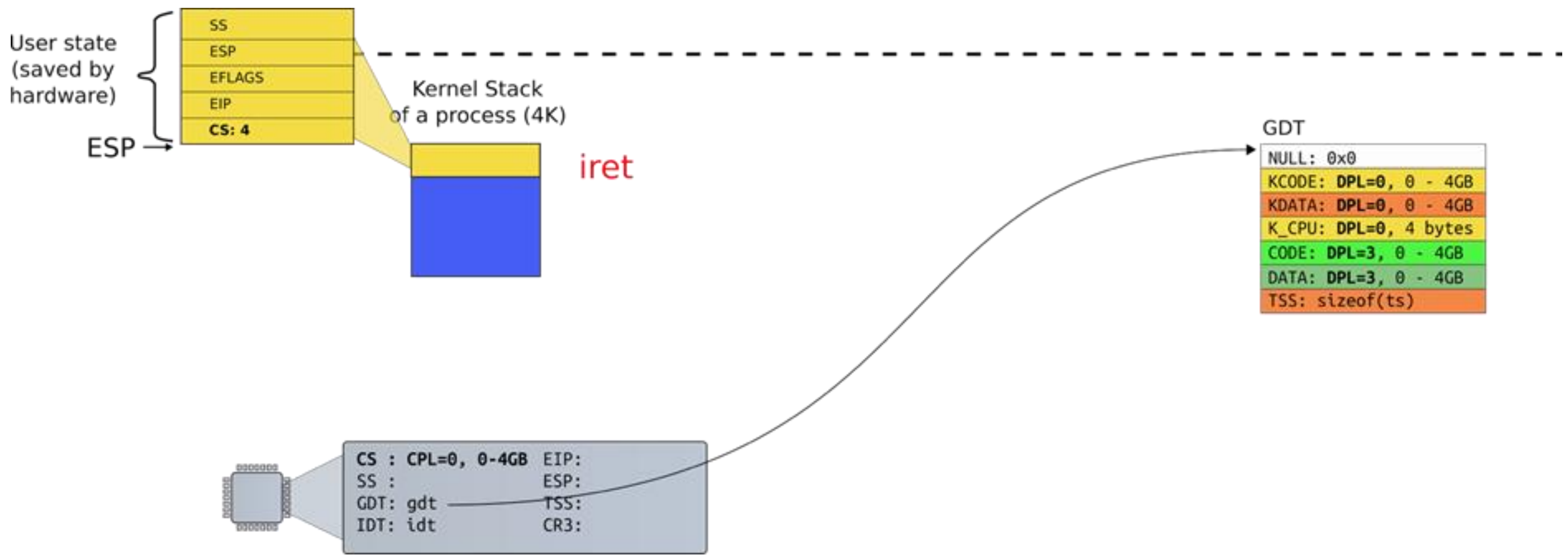
How GDT is defined

- 9180 # Bootstrap GDT
- 9181 .p2align 2 # force 4 byte alignment
- 9182 gdt:
- 9183   SEG_NULLASM # null seg
- 9184   SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff) # code seg
- 9185   SEG_ASM(STA_W, 0x0, 0xffffffff) # data seg
- 9186
- 9187 gdtdesc:
- 9188   .word (gdtdesc − gdt − 1) # sizeof(gdt) − 1
- 9189   .long gdt

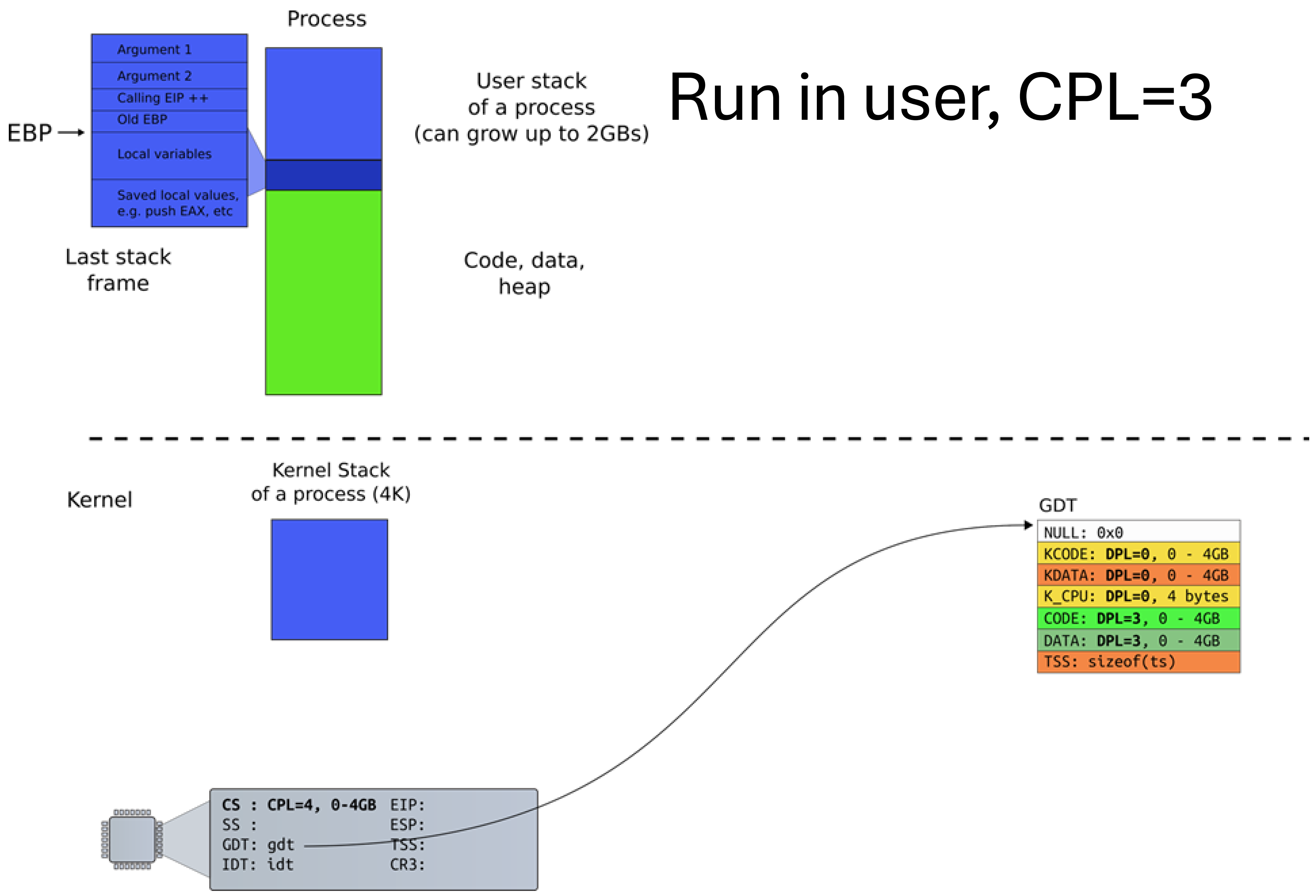
Now CPL=0. We run in the kernel



iret: return to user, load GDT #4



Run in user, CPL=3


Process
Argument 1
Argument 2
Calling EIP ++
Old EBP
Local variables
Saved local values, e.g. push EAX, etc
EBP
Last stack frame
User stack of a process (can grow up to 2GBs)
Code, data, heap
Kernel
Kernel Stack of a process (4K)
GDT

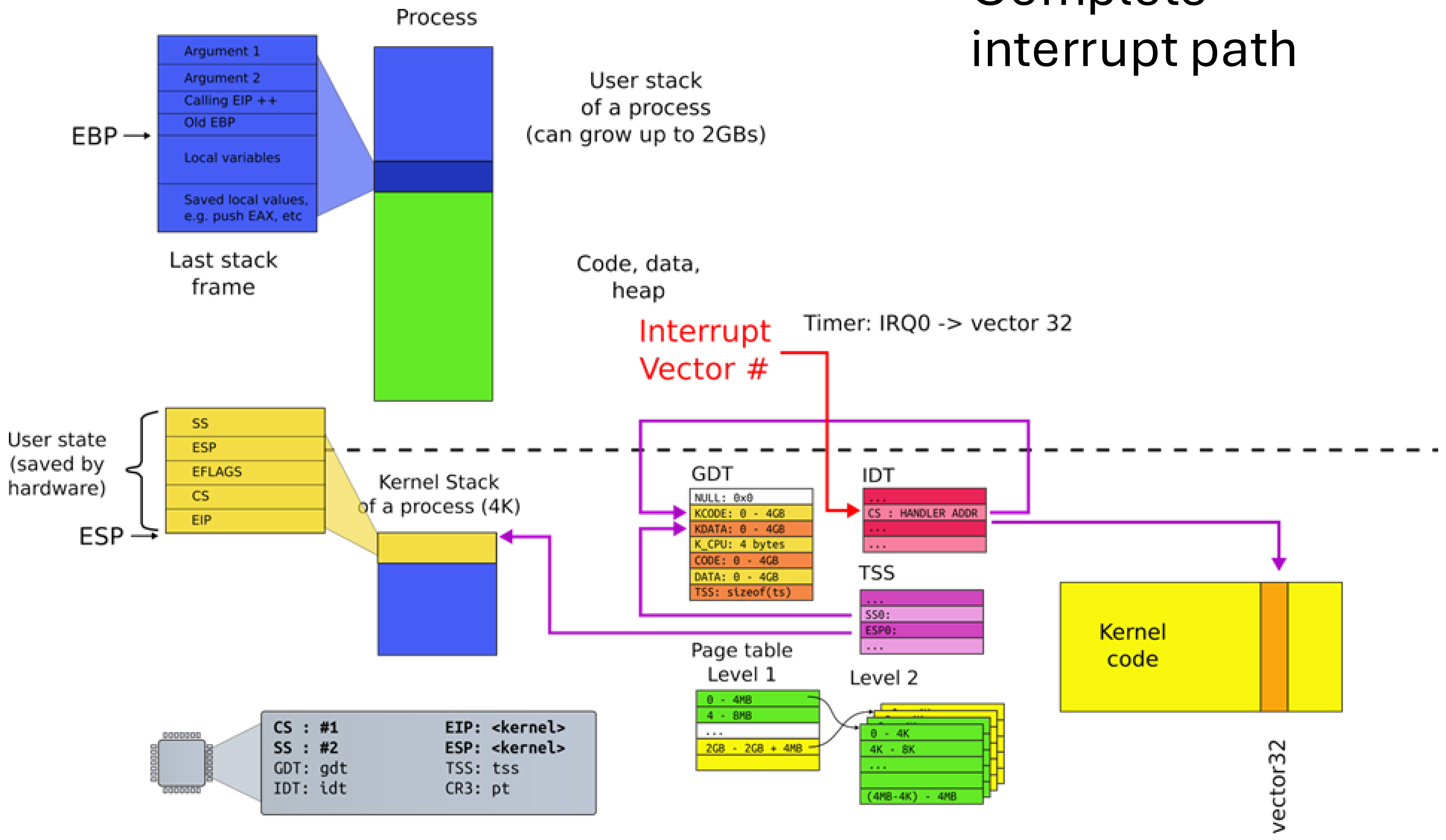
NULL: 0x0
KCODE: **DPL=0,** 0 - 4GB
KDATA: **DPL=0,** 0 - 4GB
K_CPU: **DPL=0,** 4 bytes
CODE: **DPL=3,** 0 - 4GB
DATA: **DPL=3,** 0 - 4GB
TSS: sizeof(ts)

```
CS : CPL=4, 0-4GB EIP:
SS :              ESP:
GDT: gdt          TSS:
IDT: idt          CR3:
```

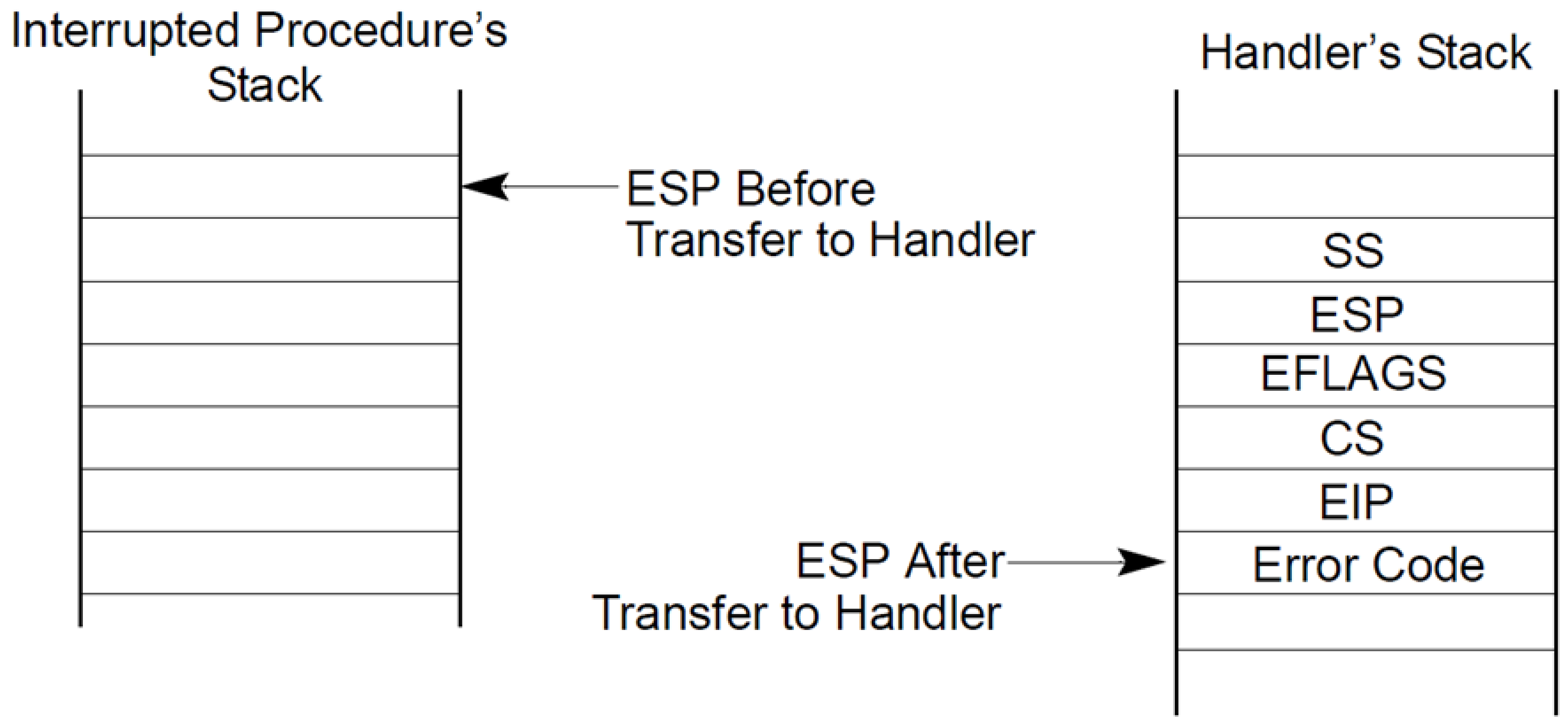
Real world

- Only two privilege levels are used in modern OSes:
  - OS kernel runs at 0
  - User code runs at 3
- This is called “flat” segment model
  - Segments for both 0 and 3 cover entire address space

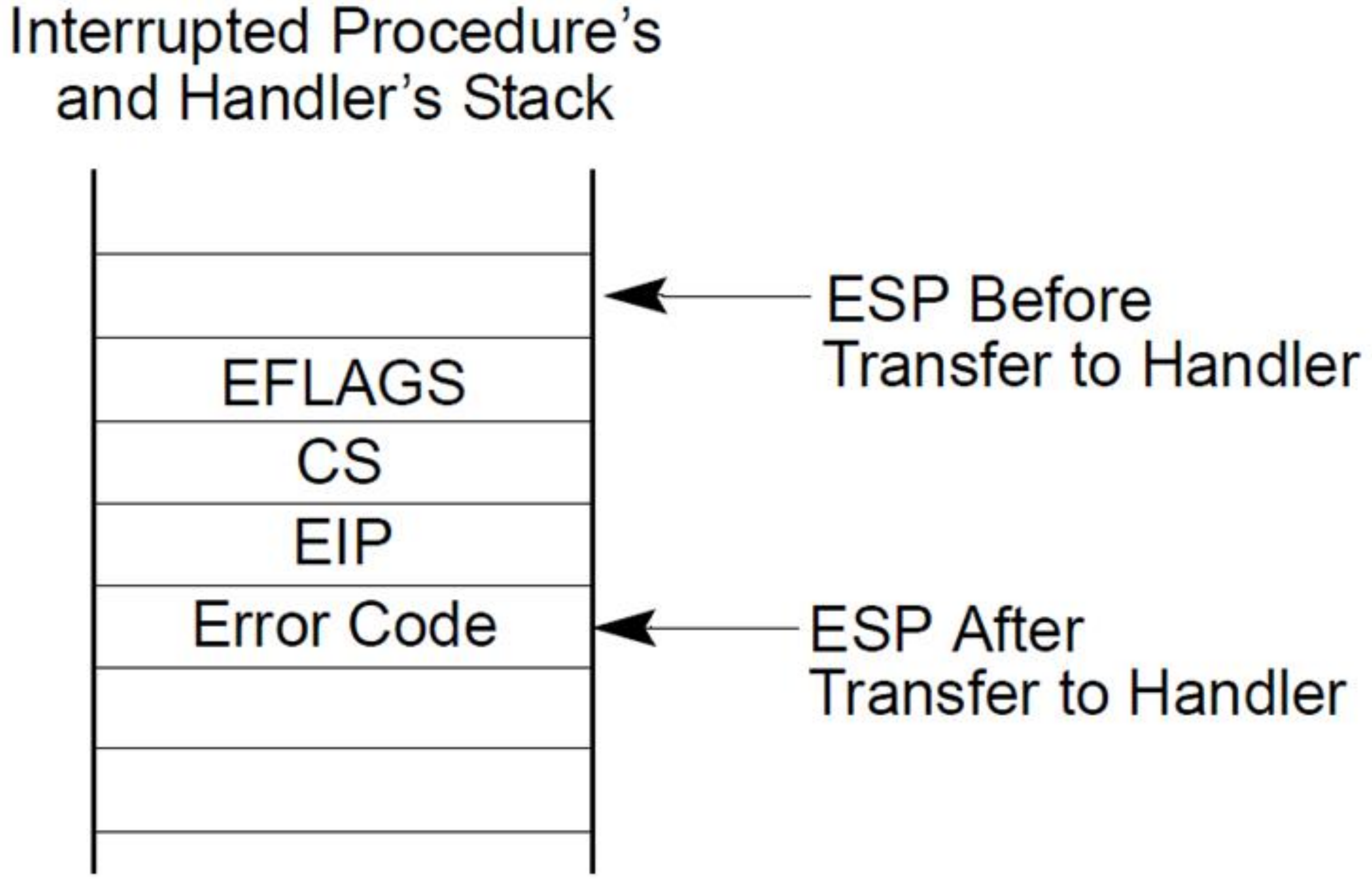
- Complete interrupt path



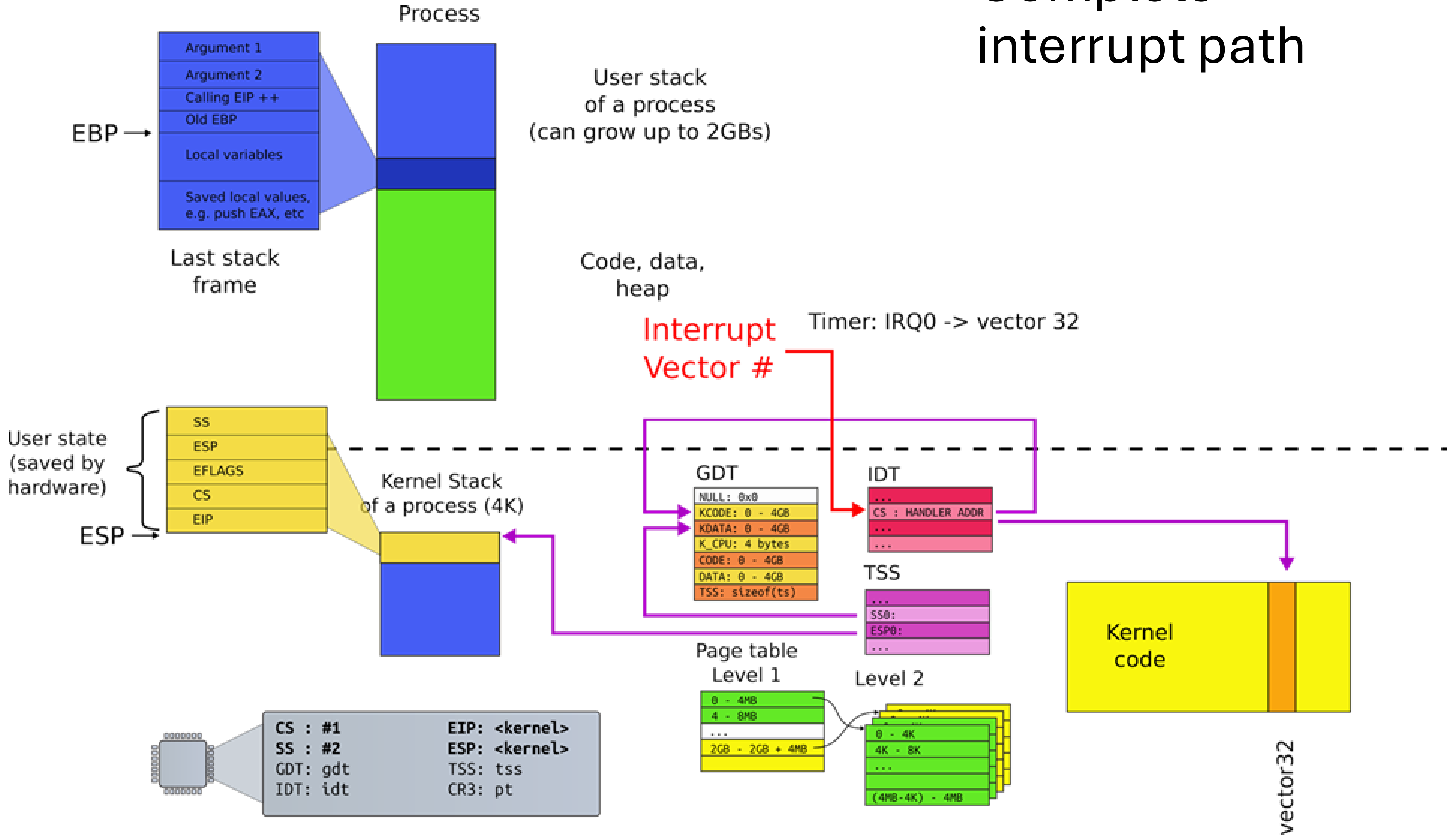
Stack Usage with Privilege-Level Change


Interrupted Procedure's Stack
Handler's Stack
ESP Before Transfer to Handler
SS
ESP
EFLAGS
CS
EIP
Error Code
ESP After Transfer to Handler

Stack Usage with No Privilege-Level Change

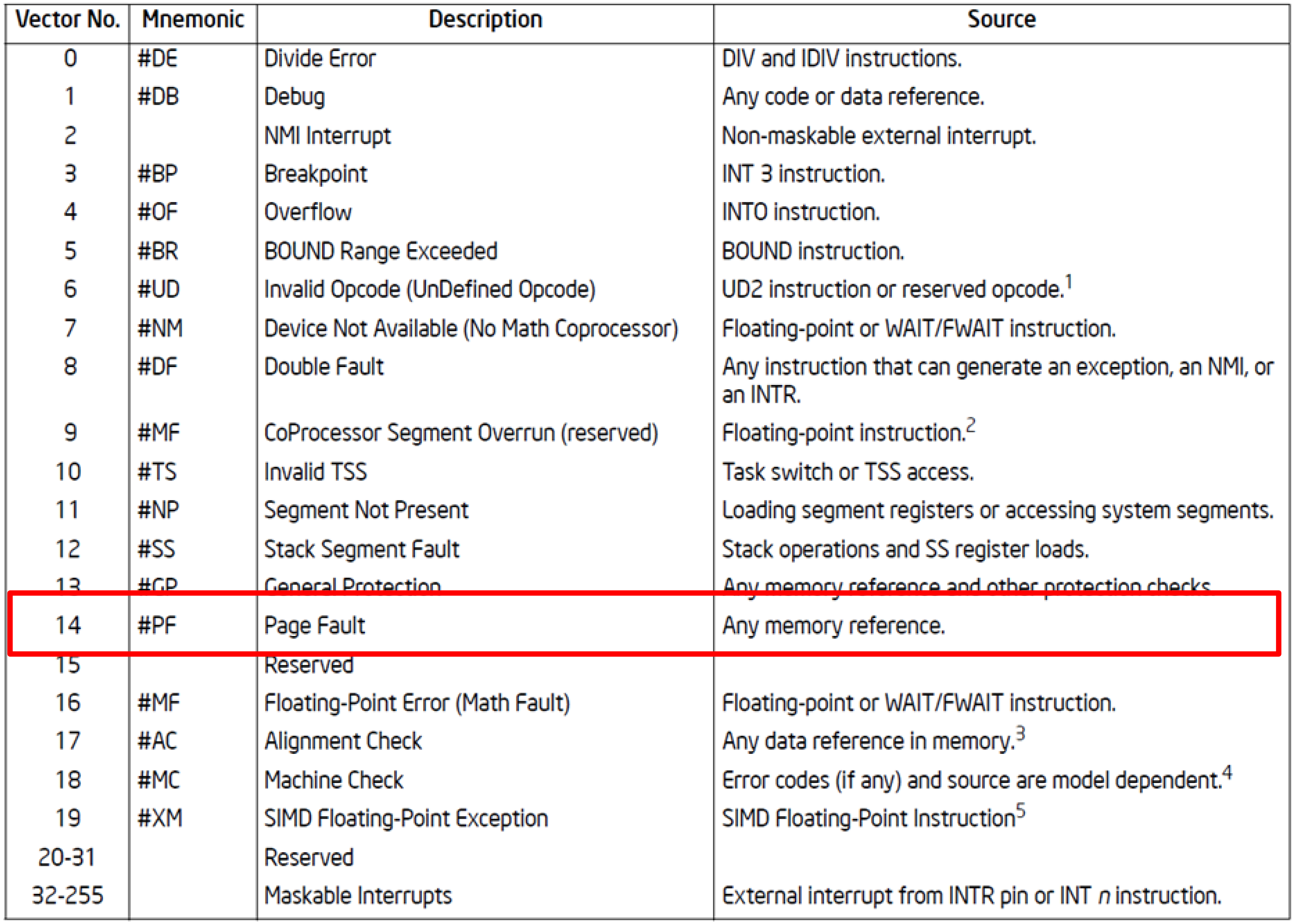

Interrupted Procedure's and Handler's Stack
EFLAGS
CS
EIP
Error Code
ESP Before Transfer to Handler
ESP After Transfer to Handler

- Complete interrupt path



- Interrupt descriptor table (IDT)

Vector No.	Mnemonic	Description	Source
0	#DE	Divide Error	DIV and IDIV instructions.
1	#DB	Debug	Any code or data reference.
2		NMI Interrupt	Non-maskable external interrupt.
3	#BP	Breakpoint	INT 3 instruction.
4	#OF	Overflow	INTO instruction.
5	#BR	BOUND Range Exceeded	BOUND instruction.
6	#UD	Invalid Opcode (UnDefined Opcode)	UD2 instruction or reserved opcode.[1]
7	#NM	Device Not Available (No Math Coprocessor)	Floating-point or WAIT/FWAIT instruction.
8	#DF	Double Fault	Any instruction that can generate an exception, an NMI, or an INTR.
9	#MF	CoProcessor Segment Overrun (reserved)	Floating-point instruction.[2]
10	#TS	Invalid TSS	Task switch or TSS access.
11	#NP	Segment Not Present	Loading segment registers or accessing system segments.
12	#SS	Stack Segment Fault	Stack operations and SS register loads.
13	#GP	General Protection	Any memory reference and other protection checks.
14	#PF	Page Fault	Any memory reference.
15		Reserved	
16	#MF	Floating-Point Error (Math Fault)	Floating-point or WAIT/FWAIT instruction.
17	#AC	Alignment Check	Any data reference in memory.[3]
18	#MC	Machine Check	Error codes (if any) and source are model dependent.[4]
19	#XM	SIMD Floating-Point Exception	SIMD Floating-Point Instruction[5]
20-31		Reserved	
32-255		Maskable Interrupts	External interrupt from INTR pin or INT *n* instruction.



Vector No.	Mnemonic	Description	Source
0	#DE	Divide Error	DIV and IDIV instructions.
1	#DB	Debug	Any code or data reference.
2		NMI Interrupt	Non-maskable external interrupt.
3	#BP	Breakpoint	INT 3 instruction.
4	#OF	Overflow	INTO instruction.
5	#BR	BOUND Range Exceeded	BOUND instruction.
6	#UD	Invalid Opcode (UnDefined Opcode)	UD2 instruction or reserved opcode.[1]
7	#NM	Device Not Available (No Math Coprocessor)	Floating-point or WAIT/FWAIT instruction.
8	#DF	Double Fault	Any instruction that can generate an exception, an NMI, or an INTR.
9	#MF	CoProcessor Segment Overrun (reserved)	Floating-point instruction.[2]
10	#TS	Invalid TSS	Task switch or TSS access.
11	#NP	Segment Not Present	Loading segment registers or accessing system segments.
12	#SS	Stack Segment Fault	Stack operations and SS register loads.
13	#GP	General Protection	Any memory reference and other protection checks.
14	#PF	Page Fault	Any memory reference.
15		Reserved	
16	#MF	Floating-Point Error (Math Fault)	Floating-point or WAIT/FWAIT instruction.
17	#AC	Alignment Check	Any data reference in memory.[3]
18	#MC	Machine Check	Error codes (if any) and source are model dependent.[4]
19	#XM	SIMD Floating-Point Exception	SIMD Floating-Point Instruction[5]
20-31		Reserved	
32-255		Maskable Interrupts	External interrupt from INTR pin or INT *n* instruction.

Interrupts

- Each type of interrupt is assigned an index from 0 - 255.
- 0 -31 are for processor interrupts fixed by Intel
    - E.g., 14 is always for page faults
- 32 - 255 are software configured
- 32 - 47 are often used for device interrupts (IRQs)
- **0x80 issues system call in Linux**
    - **Xv6 uses 0x40 (64) for the system call**

Disabling interrupts

- Delivery of interrupts can be disabled with IF (interrupt flag) in EFLAGS register
- There is a couple of exceptions
- Synchronous interrupts cannot be disabled
    - It doesn't make sense to disable a page fault
    - INT n – cannot be masked as it is synchronous
- Non-maskable interrupts (see next slide)
    - Interrupt #2 in the IDT

Vector No.	Mnemonic	Description	Source
0	#DE	Divide Error	DIV and IDIV instructions.
1	#DB	Debug	Any code or data reference.
2		NMI Interrupt	Non-maskable external interrupt.
3	#BP	Breakpoint	INT 3 instruction.
4	#OF	Overflow	INTO instruction.
5	#BR	BOUND Range Exceeded	BOUND instruction.
6	#UD	Invalid Opcode (UnDefined Opcode)	UD2 instruction or reserved opcode.[1]
7	#NM	Device Not Available (No Math Coprocessor)	Floating-point or WAIT/FWAIT instruction.
8	#DF	Double Fault	Any instruction that can generate an exception, an NMI, or an INTR.
9	#MF	CoProcessor Segment Overrun (reserved)	Floating-point instruction.[2]
10	#TS	Invalid TSS	Task switch or TSS access.
11	#NP	Segment Not Present	Loading segment registers or accessing system segments.
12	#SS	Stack Segment Fault	Stack operations and SS register loads.
13	#GP	General Protection	Any memory reference and other protection checks.
14	#PF	Page Fault	Any memory reference.
15		Reserved	
16	#MF	Floating-Point Error (Math Fault)	Floating-point or WAIT/FWAIT instruction.
17	#AC	Alignment Check	Any data reference in memory.[3]
18	#MC	Machine Check	Error codes (if any) and source are model dependent.[4]
19	#XM	SIMD Floating-Point Exception	SIMD Floating-Point Instruction[5]
20-31		Reserved	
32-255		Maskable Interrupts	External interrupt from INTR pin or INT *n* instruction.

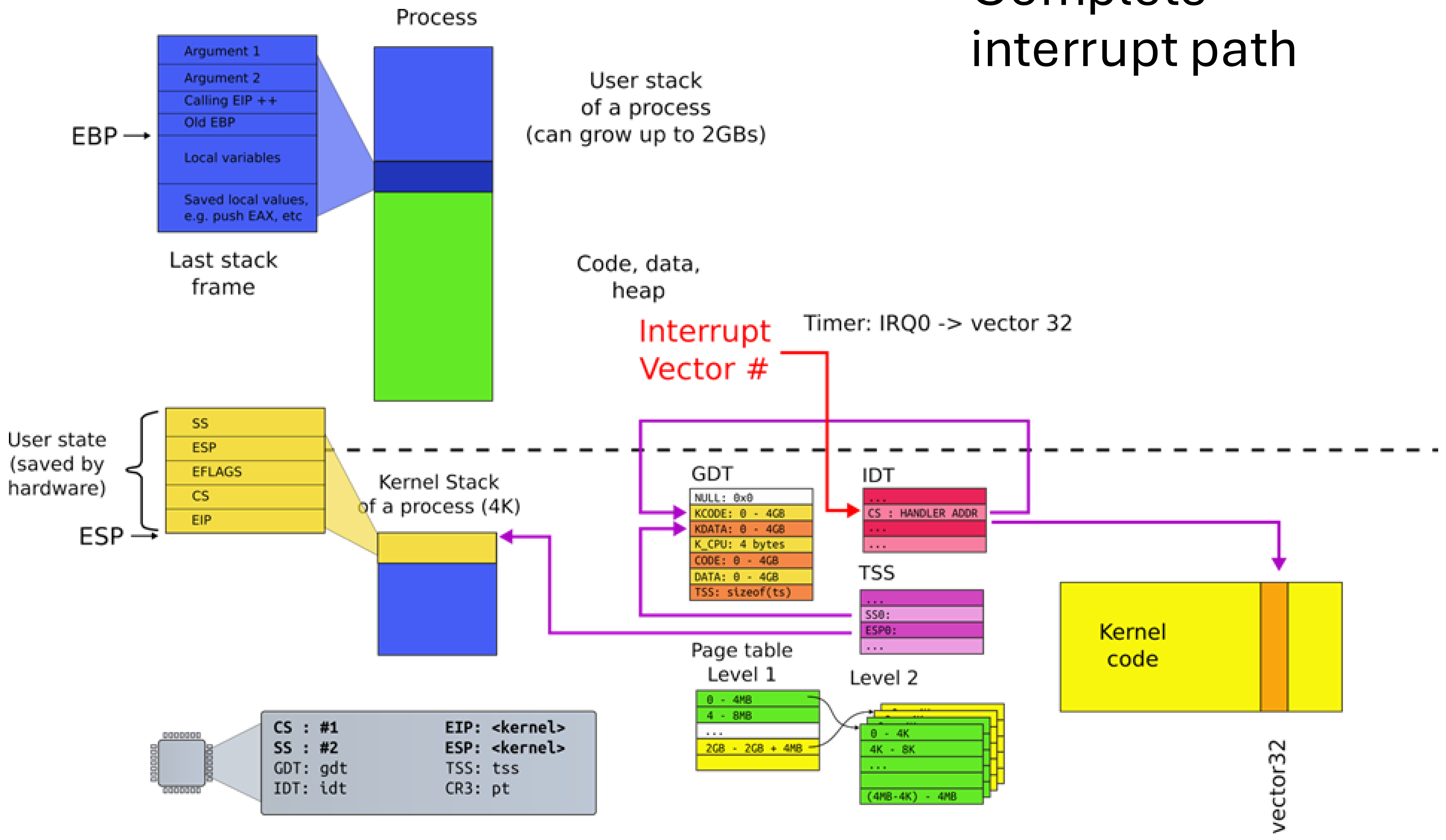
Nonmaskable interrupts (NMI)

- Delivered even if IF is clear, e.g. interrupts disabled
  - CPU blocks subsequent NMI interrupts until IRET
- Delivered via interrupt #2
  - Non-recoverable system errors
    - Chipset or memory errors
  - Trigger debugger or register dump
    - In an extremely bad state

Let’s make it concrete

Segmentation and interrupts in 64bit mode

- Complete interrupt path



We need the following

- Global Descriptor Table (GDT)
  - To configure privilege levels and change them
  - To have a pointer to TSS
- Task State Segment (TSS)
  - To have a pointer to the kernel stack
  - And some other stacks… bear with me
- Interrupt Descriptor Table
  - To actually have pointers to the interrupt handlers
- Well… a kernel stack itself
  - One per thread of execution

GDT

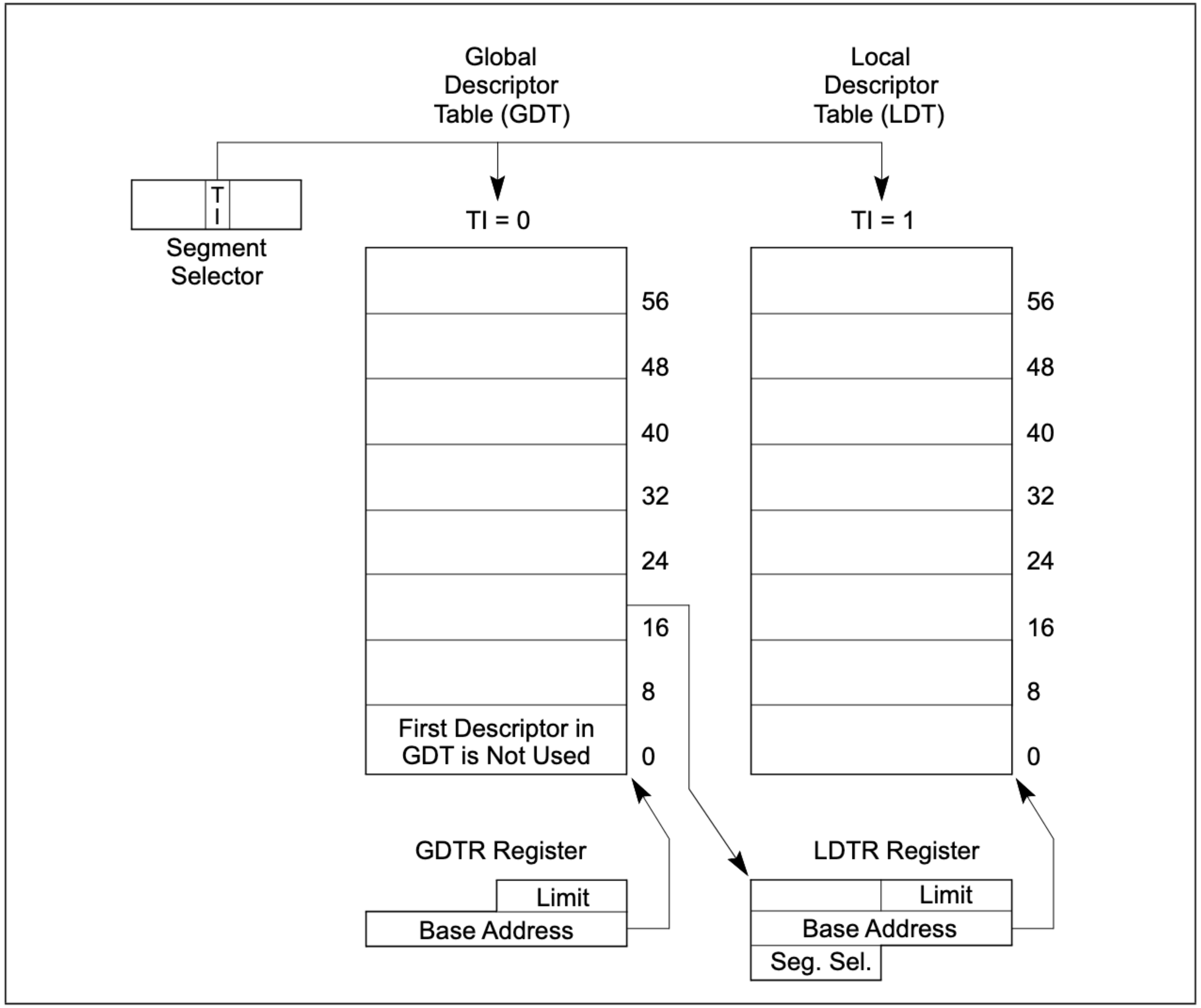
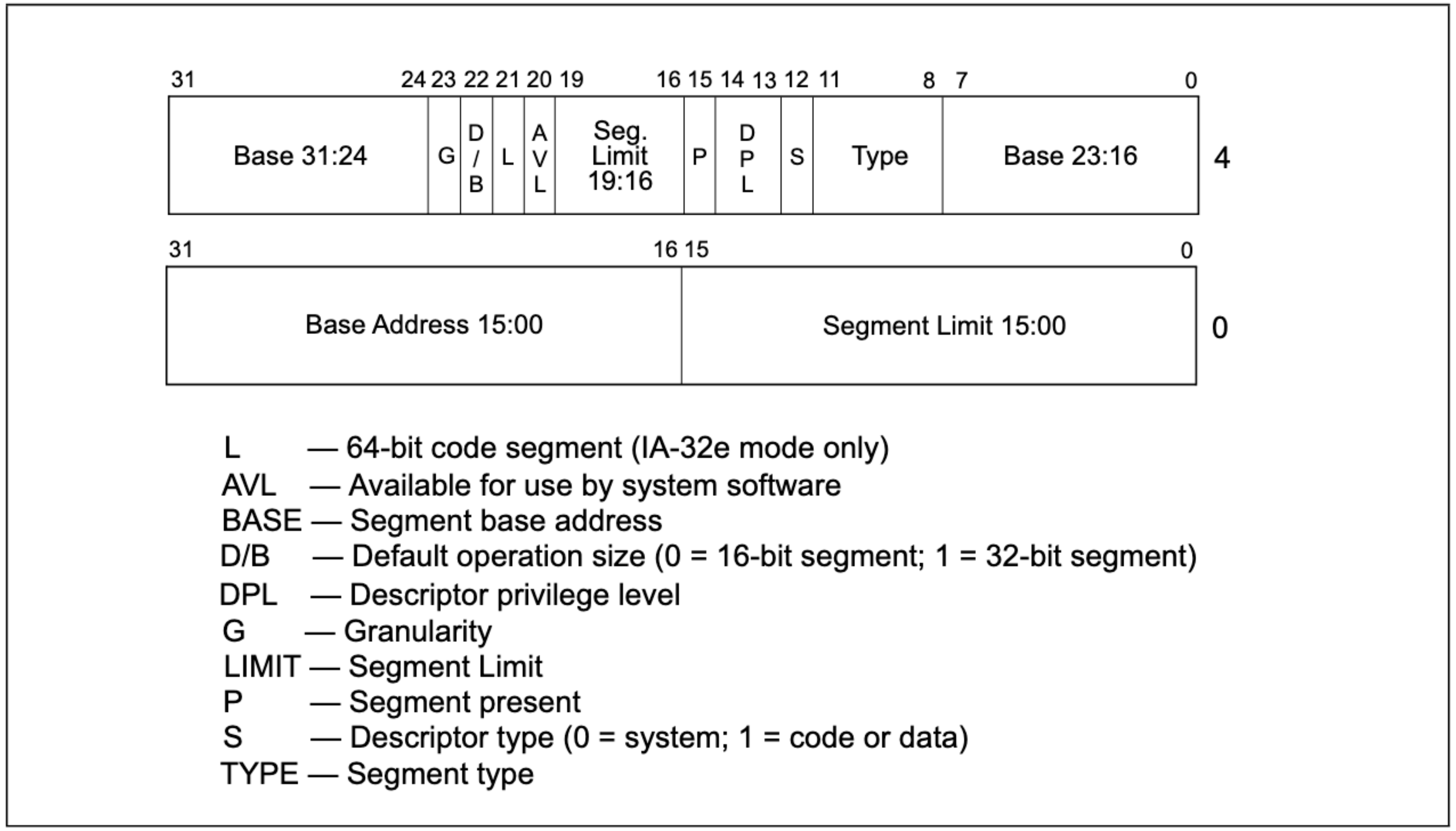


Figure 3-10. Global and Local Descriptor Tables

Segment Descriptor



L	— 64-bit code segment (IA-32e mode only)
AVL	— Available for use by system software
BASE	— Segment base address
D/B	— Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
DPL	— Descriptor privilege level
G	— Granularity
LIMIT	— Segment Limit
P	— Segment present
S	— Descriptor type (0 = system; 1 = code or data)
TYPE	— Segment type

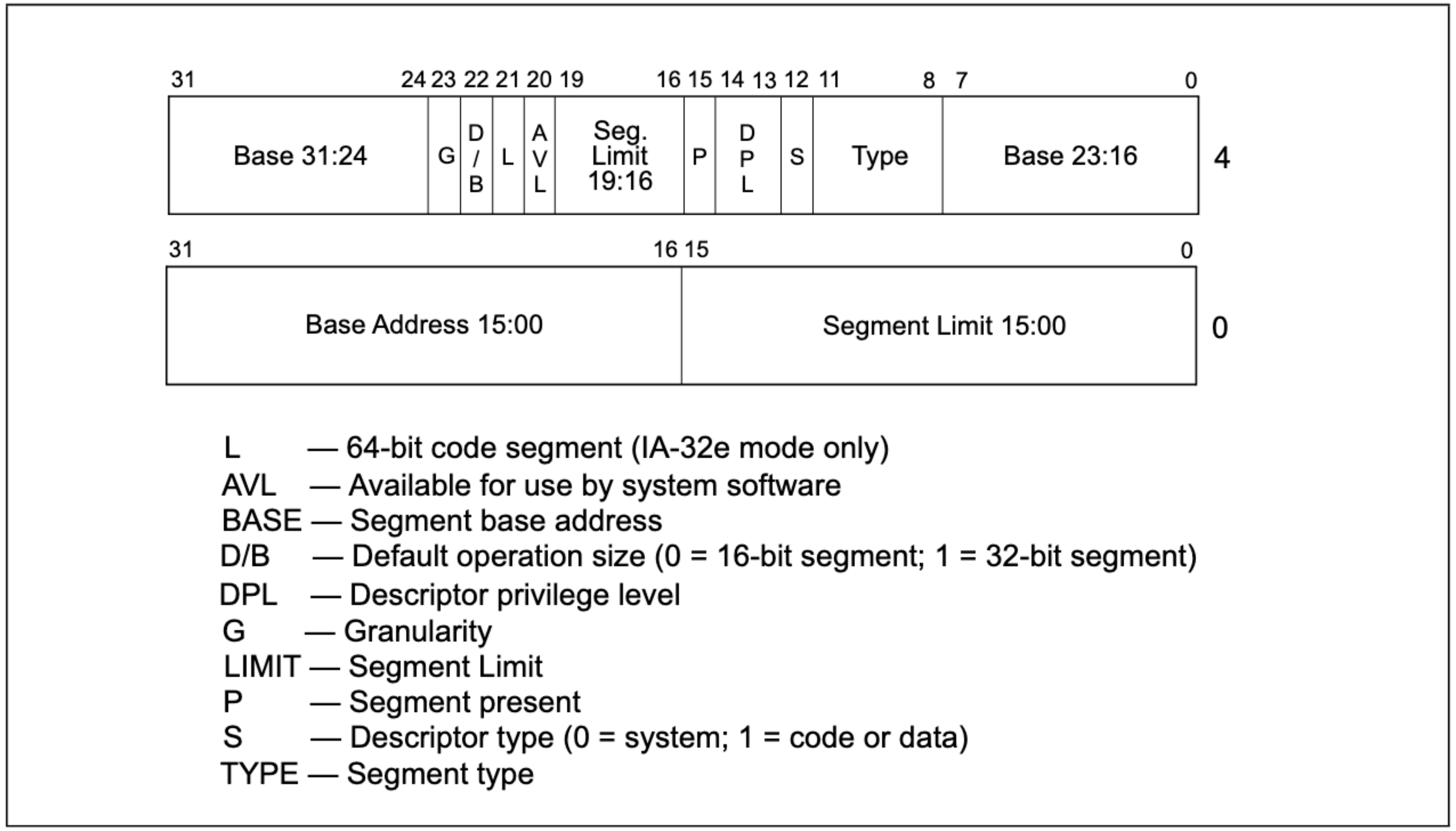
Figure 3-8. Segment Descriptor

- In 64bit mode segmentation is almost removed
- Base = 0
- Limit = MAX
- However:
  - DPL is used
  - L is used
  - Type (read/exec)

Notes

- FS and GS are the only segments that can carry non-zero base
- Not loaded from GDT
  - Their base values come from **MSRs**:
  - IA32_FS_BASE (MSR 0xC0000100)
  - IA32_GS_BASE (MSR 0xC0000101)
  - IA32_KERNEL_GS_BASE (MSR 0xC0000102, swapped in via SWAPGS).

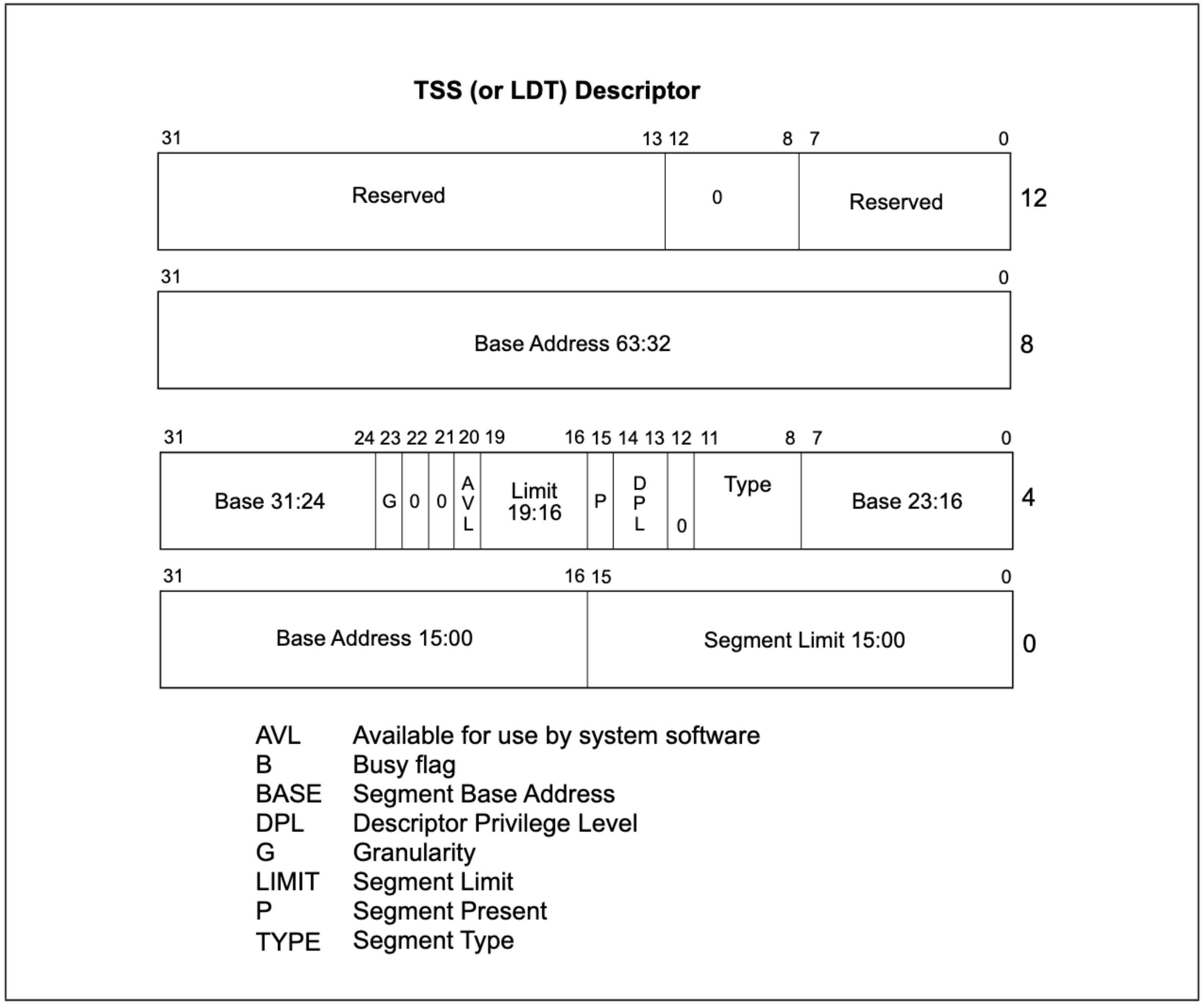
Lovely… but what about TSS?



L	— 64-bit code segment (IA-32e mode only)
AVL	— Available for use by system software
BASE	— Segment base address
D/B	— Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
DPL	— Descriptor privilege level
G	— Granularity
LIMIT	— Segment Limit
P	— Segment present
S	— Descriptor type (0 = system; 1 = code or data)
TYPE	— Segment type

Figure 3-8. Segment Descriptor

- When base and limit are ignored descriptor can fit into 64bit
- But what about TSS?
  - After all it has to point to the TSS table in memory?

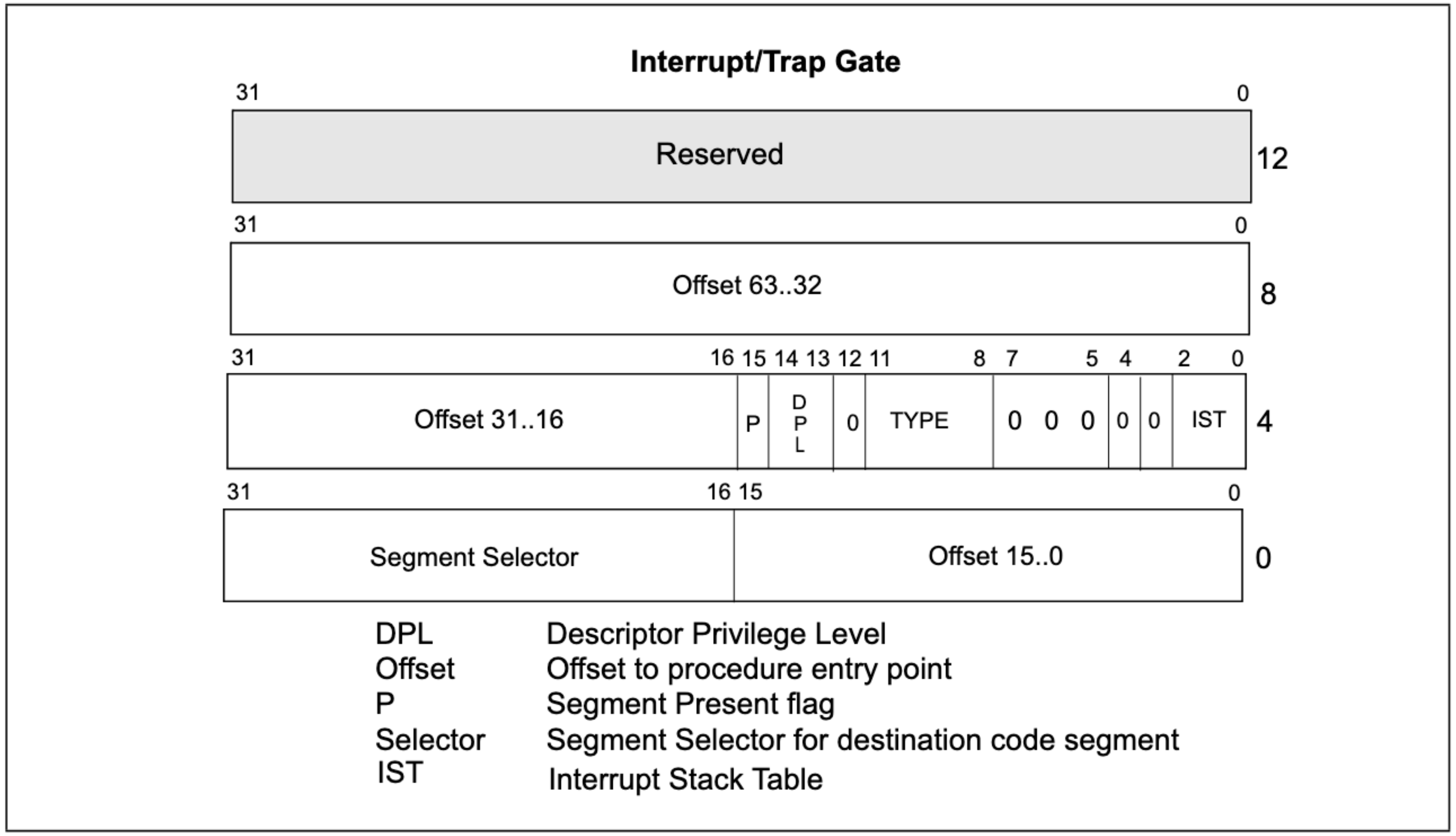


AVL	Available for use by system software
B	Busy flag
BASE	Segment Base Address
DPL	Descriptor Privilege Level
G	Granularity
LIMIT	Segment Limit
P	Segment Present
TYPE	Segment Type

Figure 9-4. Format of TSS and LDT Descriptors in 64-bit Mode

TSS is special

- 128bit descriptor



DPL	Descriptor Privilege Level
Offset	Offset to procedure entry point
P	Segment Present flag
Selector	Segment Selector for destination code segment
IST	Interrupt Stack Table

Figure 7-8. 64-Bit IDT Gate Descriptors

TSS is special

- 128bit descriptor

GDT

- Minimal GDT

```
/// A Global Descriptor Table.
#[derive(Debug)]
#[repr(packed)]
pub struct GlobalDescriptorTable {
/// Null entry.
_null: GdtEntry,

/// Kernel data.
pub kernel_data: GdtEntry,

/// Kernel code.
pub kernel_code: GdtEntry,

/// User data.
pub user_data: GdtEntry,

/// User code.
pub user_code: GdtEntry,

/// TSS.
///
/// This is 16 bytes in Long Mode.
pub tss: BigGdtEntry,
}
```

```rust
/// A 8-byte GDT Code/Data entry.
#[derive(Copy, Clone, Debug)]
#[repr(packed)]
#[allow(dead_code)] // "field is never read" - used by platform
pub struct GdtEntry {
    limitl: u16,
    offsetl: u16,
    offsetm: u8,
    access: u8,
    flags_limith: u8,
    offseth: u8,
}
```

Regular GDT entry

Big GDT entry for TSS

```rust
/// A 16-byte GDT System entry.
///
/// This is described in Figure 4-22 in AMD Vol. 2.
#[derive(Copy, Clone, Debug)]
#[repr(packed)]
#[allow(dead_code)] // "field is never read" - used by platform
pub struct BigGdtEntry {
    limitl: u16,
    offsetl: u16,
    offsetm: u8,
    access_type: u8,
    flags_limith: u8,
    offseth: u8,
    offsetx: u32,
    _reserved: u32,
}
```

Enabling external interrupts

- Legacy PIC
- Modern setup: LAPIC + I/O APIC

Legacy PIC

- See Phillipp’s blog for details

LAPIC and I/O APIC

Local APIC (per CPU core)

- Implemented inside the CPU core
- Handles:
  - Core-local interrupts (timer, thermal, performance counter, error, LINT0/1 pins)
  - Receiving and prioritizing interrupts delivered *to that CPU*
  - Sending inter-processor interrupts (IPIs) to other cores
  - Knows *which IDT vector* to invoke based on its **Local Vector Table (LVT)**
- **But** it doesn’t know anything about *external hardware devices* (like NIC, disk, keyboard).

I/O APIC handles external hardware interrupts (from devices)

- Devices (keyboard, NIC, disk, etc.) assert an IRQ line
- The **I/O APIC** routes that IRQ to a LAPIC, by sending an **APIC interrupt message** (with vector, delivery mode, destination, etc.)
- The LAPIC receives it and delivers the specified **interrupt vector** to its CPU
- Example: keyboard IRQ1 → I/O APIC pin 1 → LAPIC → vector 0x21

Let’s look at LAPIC sources one more time

- Each CPU core has a **Local APIC**.
- The LAPIC can receive interrupts from several sources:
  - The **I/O APIC** (via the APIC bus or xAPIC registers)
  - **Inter-processor interrupts (IPIs)** from other cores
  - **Local sources** like timer, thermal sensor, performance counters
  - And finally: **two external pins, LINT0 and LINT1**

Local interrupts (generated by LAPIC itself)

- The LAPIC can generate interrupts internally for CPU management:
  - **Timer interrupt** (programmable per-CPU timer in LAPIC, replaces PIT/HPET in SMP systems).
  - **LINT0/LINT1 pins**: can be programmed for:
    - Legacy INTR (8259 PIC passthrough, if enabled)
    - NMI (Non-Maskable Interrupt)
    - SMI (System Management Interrupt, usually routed elsewhere)
    - External interrupts for special routing
  - **Performance monitoring interrupts (PMI)** — from performance counters
  - **Thermal sensor interrupt** — from CPU thermal events
  - **Error interrupt** — APIC internal error signaling

Inter-processor interrupts (IPIs)

- CPUs can send interrupts to each other via LAPIC → LAPIC messaging over the APIC bus
- Used for:
  - TLB shootdown
  - Rescheduling
  - Startup (INIT/SIPI sequence for booting APs)

Spurious interrupts

- If the LAPIC is enabled, it must be given a **Spurious Interrupt Vector** in the **Spurious Interrupt Vector Register (SVR)**
- If the LAPIC receives an interrupt it can’t map properly, it raises this spurious interrupt on the CPU
- In the past it was due to electrically noisy ISA **(Industry Standard Architecture)** bus
  - Each IRQ was literally a **dedicated physical wire** between the device slot and the 8259 PIC
- Today it’s typically due to misconfiguration, disabled LAPIC, illegal vector, etc.
  - On PCI/PCIe systems, interrupts are message-based (MSI/MSI-X) or well-buffered, so spurious electrical blips are almost nonexistent
- LAPIC can deliver an interrupt using the **vector stored in SVR**

LINT0/1 Pins

- Think of LINT0/1 as **compatibility hooks**:
  - In the old days (8259 PIC only), the CPU only knew about two pins:
    - **INTR#** (maskable interrupts)
    - **NMI#** (non-maskable).
- When the LAPIC was introduced, it still had to support that wiring model, so each LAPIC exposes **two equivalent pins** that can be mapped into its internal logic

LINT0 (Local Interrupt 0)

- Usually wired to the system's **INTR# signal** (legacy 8259 PIC output)
- If the system is running in *Virtual Wire Mode* (APIC not fully used), this is how the old PIC delivers interrupts to the CPU
- When the APIC is fully enabled, LINT0 can be **programmed** in the LAPIC's **Local Vector Table (LVT)** entry for LINT0:
  - Delivery mode (Fixed, NMI, ExtINT, SMI)
  - Vector (if applicable)
  - Mask/unmask

LINT1 (Local Interrupt 1)

- Usually wired to the system’s **NMI# signal**
- By default, this delivers **non-maskable interrupts** to the CPU
- Also configurable via the **LVT LINT1 entry**. For example, you could change it to deliver a normal vector instead of NMI (not common)

I/O APIC

I/O APIC (chipset / system logic)

- **Implemented outside of the CPU**
- Provides input pins (IRQs) for devices
- Each input pin is mapped through a **Redirection Table entry**, which specifies:
  - The vector number (IDT entry).
  - The destination LAPIC (which CPU should handle it).
  - Delivery mode (fixed, lowest priority, NMI, etc.).
- **But** it cannot deliver interrupts directly to the CPU — it always forwards them to a LAPIC.

- If you only configure the **LAPIC**, you can get *local events* like the LAPIC timer working, but devices (keyboard, disk, NIC) won’t generate interrupts — because the LAPIC doesn’t see external IRQ lines
- If you only configure the **I/O APIC**, devices might assert their pins, but nothing will happen — because the LAPIC wouldn’t know what to do with the messages unless you set up its vector tables and enable it (via the Spurious Interrupt Vector Register, etc.)

Example: Keyboard key pressed → IRQ line asserted.

- Keyboard drives **IRQ1** line into the **I/O APIC**.
- I/O APIC looks up Redirection Table entry for pin 1:
    - Says: “deliver vector 0x21 to LAPIC with ID=0”
- I/O APIC sends an *Interrupt Message* on the system bus to LAPIC 0
- LAPIC 0 sees the message, looks at its LVT, delivers **vector 0x21** to the CPU
- CPU does an interrupt entry into IDT[0x21] → your handler runs

Mapping between devices and I/O APIC inputs

- The mapping from device → I/O APIC **input pin** is described by the **ACPI tables**, usually the **MADT (Multiple APIC Description Table)**
- For PCI devices:
  - Each device has an **INTx# pin** (INTA–INTD)
  - The firmware (BIOS/UEFI) or ACPI defines which **I/O APIC input pin** each device interrupt is routed to
  - Some systems allow the mapping to be programmable (PCI interrupt routing) — so it's **not strictly fixed**
- For legacy devices (keyboard, PS/2 mouse), many boards still wire them to the **same I/O APIC pins as their old IRQs**, so in practice they appear fixed

Delivery mode

- In the **I/O APIC Redirection Table (RDT) entry**, there's a **Destination Mode** and **Destination Field**:
- **Destination Mode (bit 11)**:
  - 0 → **Physical mode**: the **APIC ID** in the destination field identifies exactly one CPU.
  - 1 → **Logical mode**: the **destination field** is a **bitmask of CPUs** in the logical cluster.
- **Destination Field (upper 8 bits of the entry)**:
  - Physical mode: LAPIC ID of the target CPU.
  - Logical mode: bitmask of all CPUs in the logical APIC cluster.

Delivery Modes

- **Fixed**: deliver to the CPU(s) in the destination field normally.
- **Lowest Priority**: the I/O APIC will deliver the interrupt to the **CPU with the lowest priority** among those specified in the destination field.
- “Lowest Priority” is the key for **round-robin / load-balanced delivery**. The LAPIC has a priority register, so the next interrupt is sent to the CPU with the currently lowest priority. Over time, this achieves a form of round-robin distribution across multiple cores.

How to Configure Round-Robin

- **Set the Redirection Table entry** for the device IRQ (e.g., network card, disk) like this:
  - Delivery Mode = **Lowest Priority**
  - Destination Mode = **Logical**
  - Destination Field = **mask of all CPUs that should handle the interrupt**
- The I/O APIC then automatically distributes interrupts across the CPUs in the mask according to the LAPIC priority.

Non maskable interrupts

- **Non-maskable**: The CPU will always recognize NMIs, regardless of the state of IF (the interrupt-enable flag in RFLAGS)
  - Normal IRQs are masked when IF=0
  - NMIs *ignore* IF
- **High priority**: NMIs preempt nearly all other interrupts and exceptions, except for machine-check and double fault
- **Vector**: NMIs are always delivered at **vector 2** in the IDT (hardwired by the CPU). You cannot remap them

Use cases

- **Hardware error reporting**: e.g., ECC/parity errors in memory
- **Watchdog timers**: to break into the OS if it hangs
  - I.e., an infinite loop in a third-party device driver
- **System management / debugging**: used by some debuggers to halt CPUs
- **Reliability / availability**: some platforms use NMIs to recover from severe conditions (thermal events, etc.)

How NMIs are delivered

- On older systems, an NMI could be triggered by external hardware (e.g., memory parity error line, serious bus faults)
- In an APIC system:
  - The **LAPIC's LINT1 pin** can be configured to deliver an NMI instead of a regular interrupt
  - The I/O APIC can route an input pin to LAPIC in **NMI delivery mode**, causing an NMI when asserted
- So, NMIs can come from:
  - Motherboard hardware error signals
  - Watchdog timers (e.g., APIC timer can be set to fire an NMI for debugging)
  - I/O APIC with delivery mode set to **NMI**

Reentrancy

- When the CPU takes an **NMI**, it pushes state and jumps to the handler at IDT[2]
- **Further NMIs are blocked while the handler is running** — *but only until the CPU executes an IRET*
- This is a special case: unlike maskable IRQs (which can be reenabled early with sti), you cannot reenable NMIs inside the handler
- **At most one NMI is active at a time per CPU**

You can configure anything as NMI

- Example: keyboard
- On an xAPIC system, you control interrupt delivery using the **I/O APIC Redirection Table entry** for the keyboard's input line (usually IRQ1 → IOAPIC pin 1)
- In the **Redirection Table (lower DWORD)**:
    - **Bits 8–10 (Delivery Mode)**: set to 100b → **NMI**
    - **Bits 0–7 (Vector)**: ignored for NMI delivery. NMIs always go to **IDT[2]**.
    - Other fields (polarity, trigger mode, mask) still apply
- So yes, the I/O APIC can deliver *any external line* as an NMI by programming its entry that way

How to configure I/O APIC and LAPIC

Disable the legacy PIC

- The 8259 PIC must be disabled so it doesn’t generate spurious IRQs
- Usual method: initialize it, then mask all IRQs (write 0xFF to the IMR), or put it in a special mode (Virtual Wire → ExtINT via LINT0)

Enable the Local APIC

- **Read IA32_APIC_BASE MSR (0x1B)**
  - Set bit 11 = 1 → enable LAPIC
  - Ensure base address (usually 0xFEE00000)
- **Enable in the Spurious Interrupt Vector Register (SVR)**
  - Write your chosen spurious vector (must be ≥0x10)
  - Set bit 8 = 1 (APIC software enable)

Initialize the I/O APIC

- I/O APIC is memory-mapped (base from ACPI MADT table or MP table)
- It has two registers: **IOREGSEL** and **IOWIN**
- For each IRQ input pin (0..23 typically):
  - Program the **Redirection Table entry**:
    - **Vector**: interrupt vector number you want (≥0x20 recommended, to avoid exceptions)
    - **Delivery mode**: Fixed
    - **Destination**: target LAPIC ID (or broadcast / lowest-priority if you want round-robin)
    - **Mask**: 0 (unmask)

Thank you!

Are you rewriting our kernel in Rust?
Nah... let's just verify it