ICS143A: Principles of Operating Systems

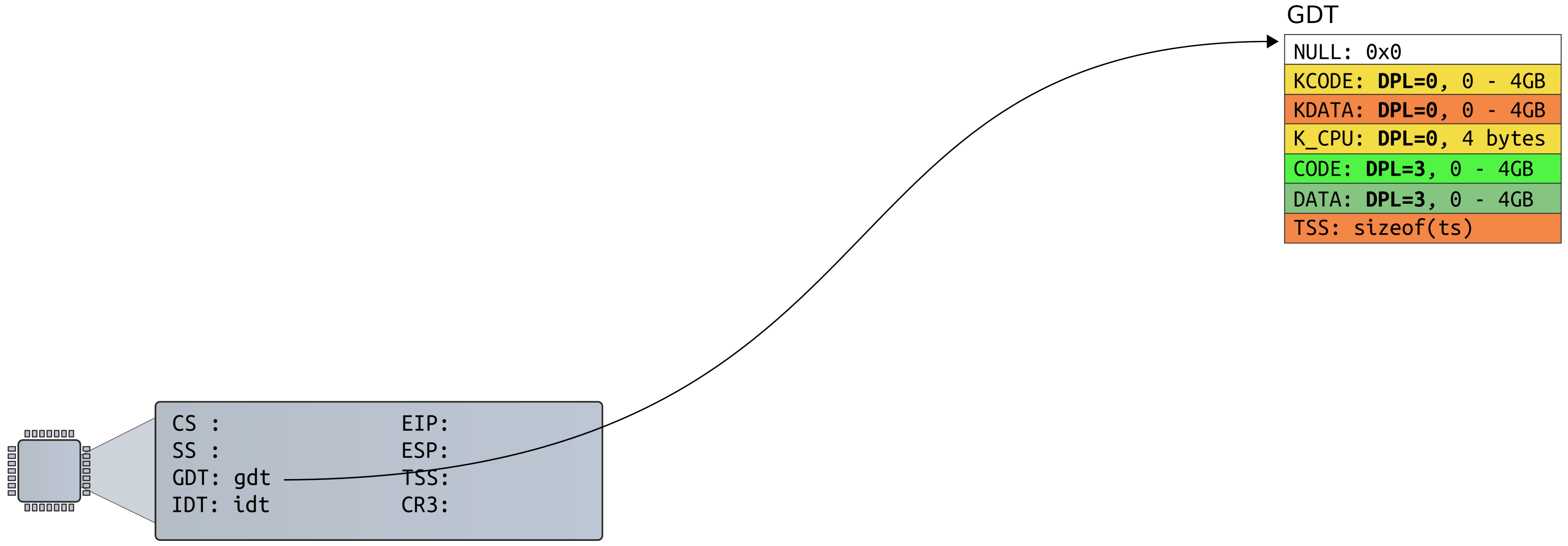
Lecture 12: Interrupts and Exceptions (part 2)

Anton Burtsev

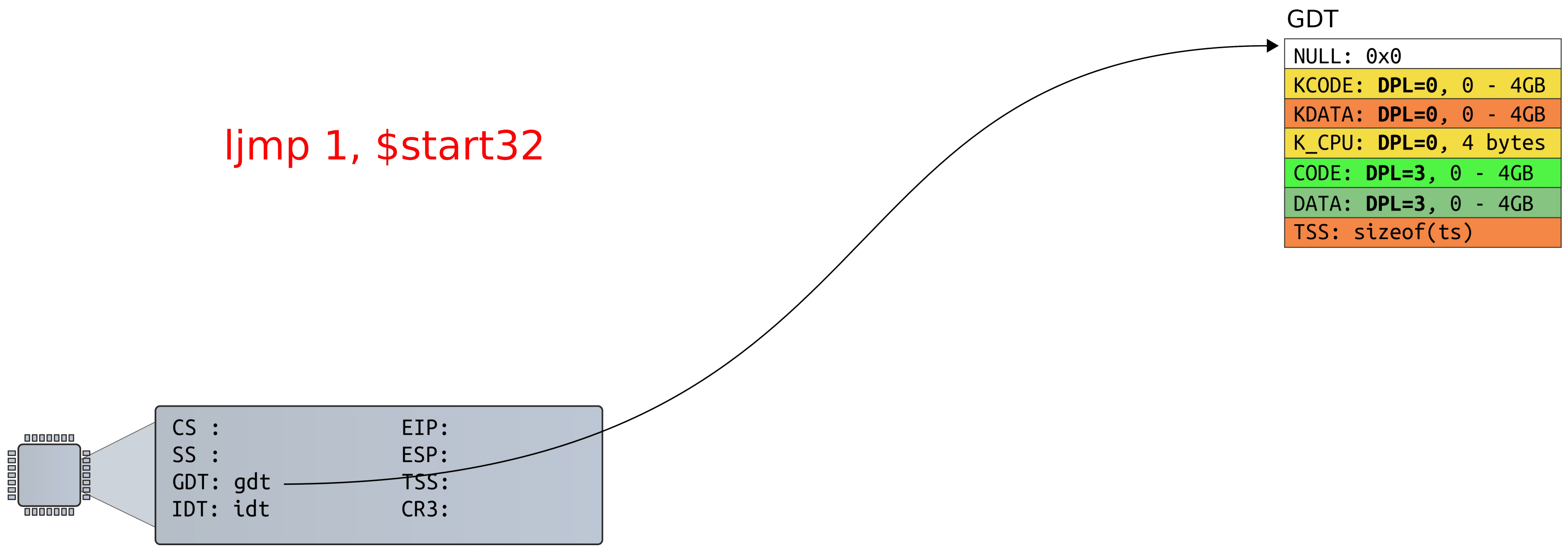
November, 2017

Privilege levels again

Started boot: no CPL yet

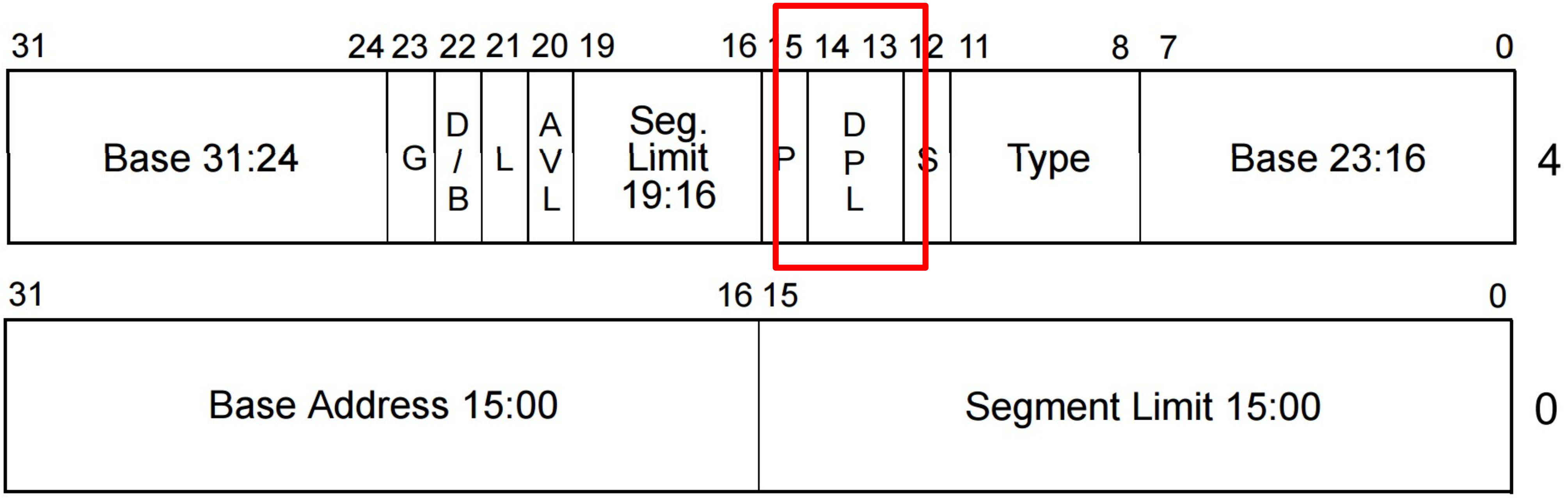


Prepare to load GDT entry #1

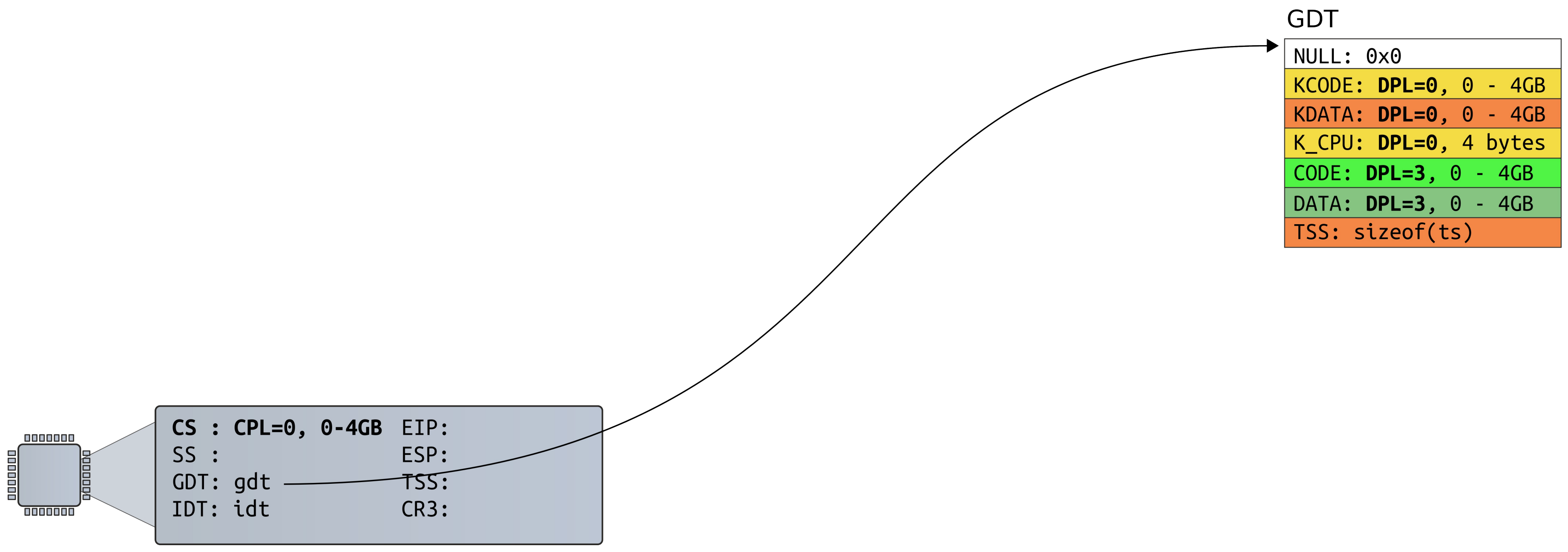


Privilege levels

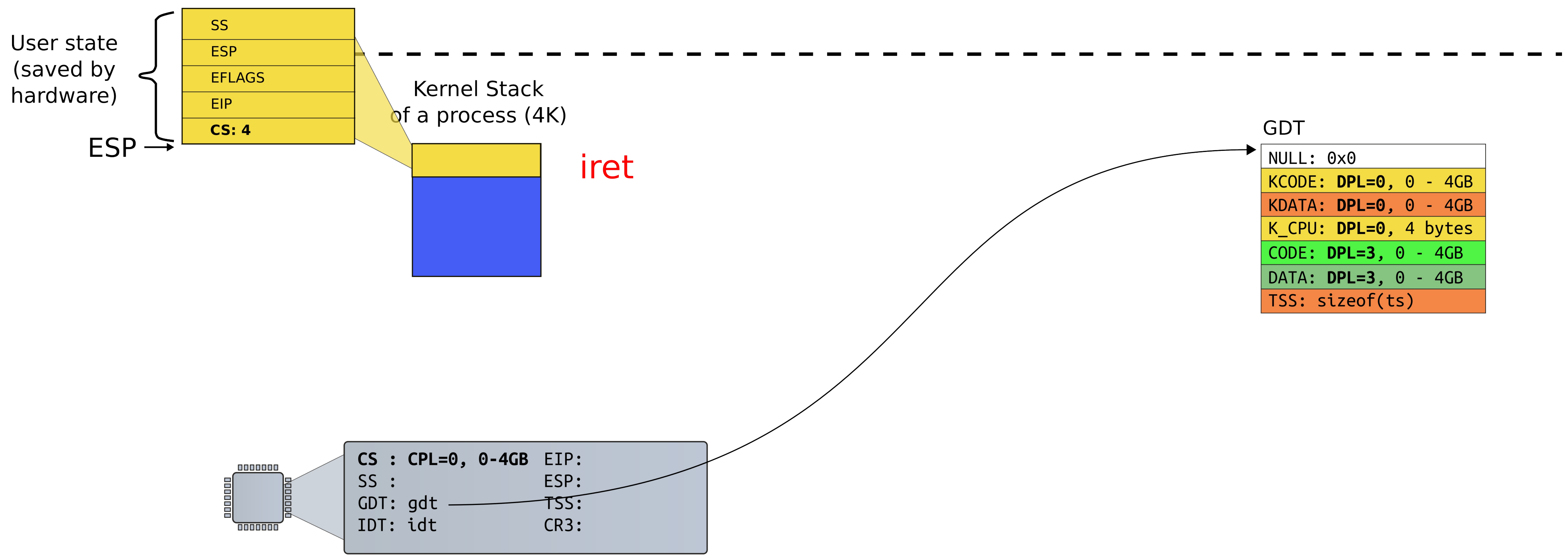
- Each segment has a privilege level
  - DPL (descriptor privilege level)
  - 4 privilege levels ranging 0-3



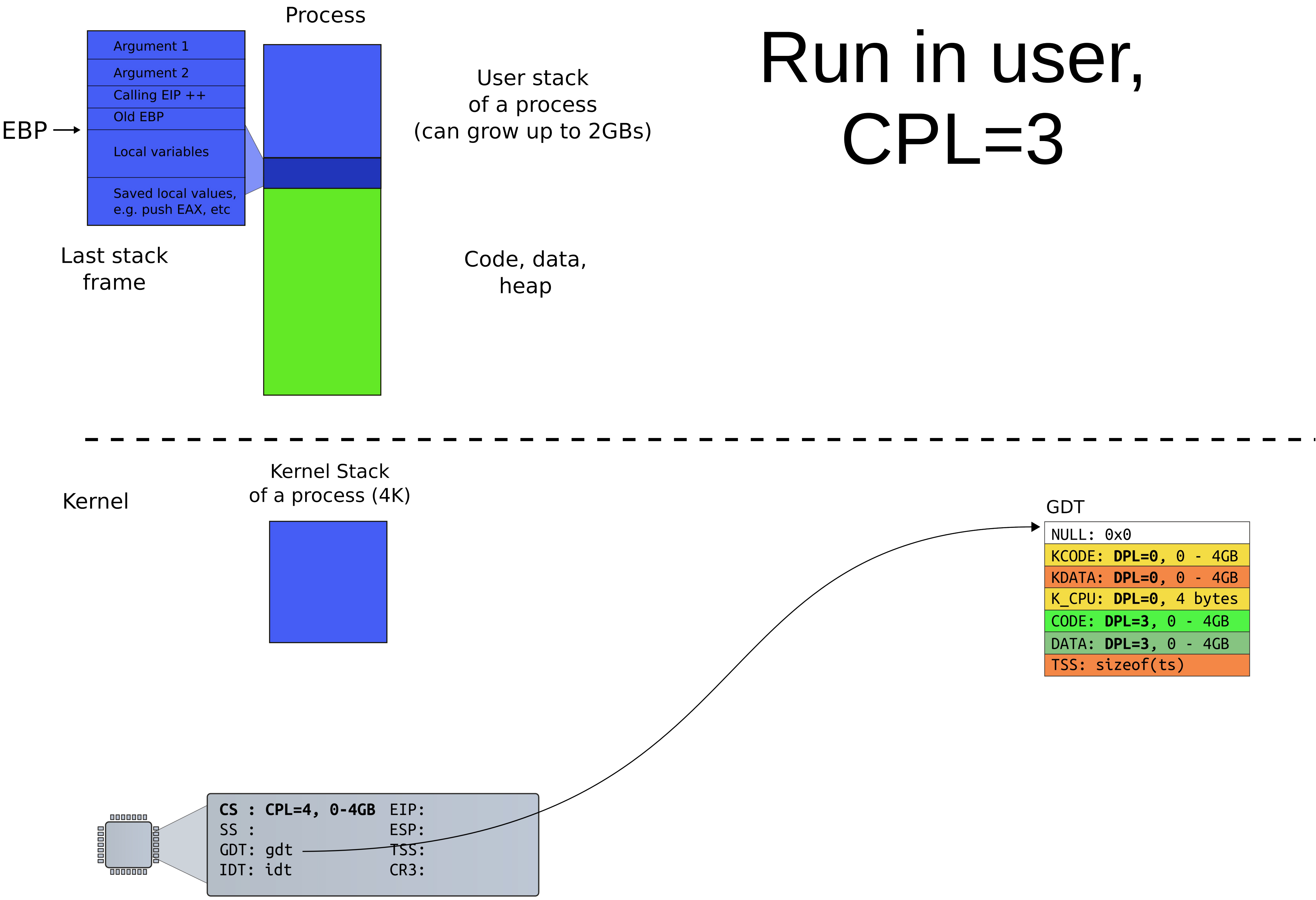
Now CPL=0. We run in the kernel



iret: return to user, load GDT #4



Run in user, CPL=3


Process
Argument 1
Argument 2
Calling EIP ++
Old EBP
EBP
Local variables
Saved local values, e.g. push EAX, etc
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:
```

Privilege levels

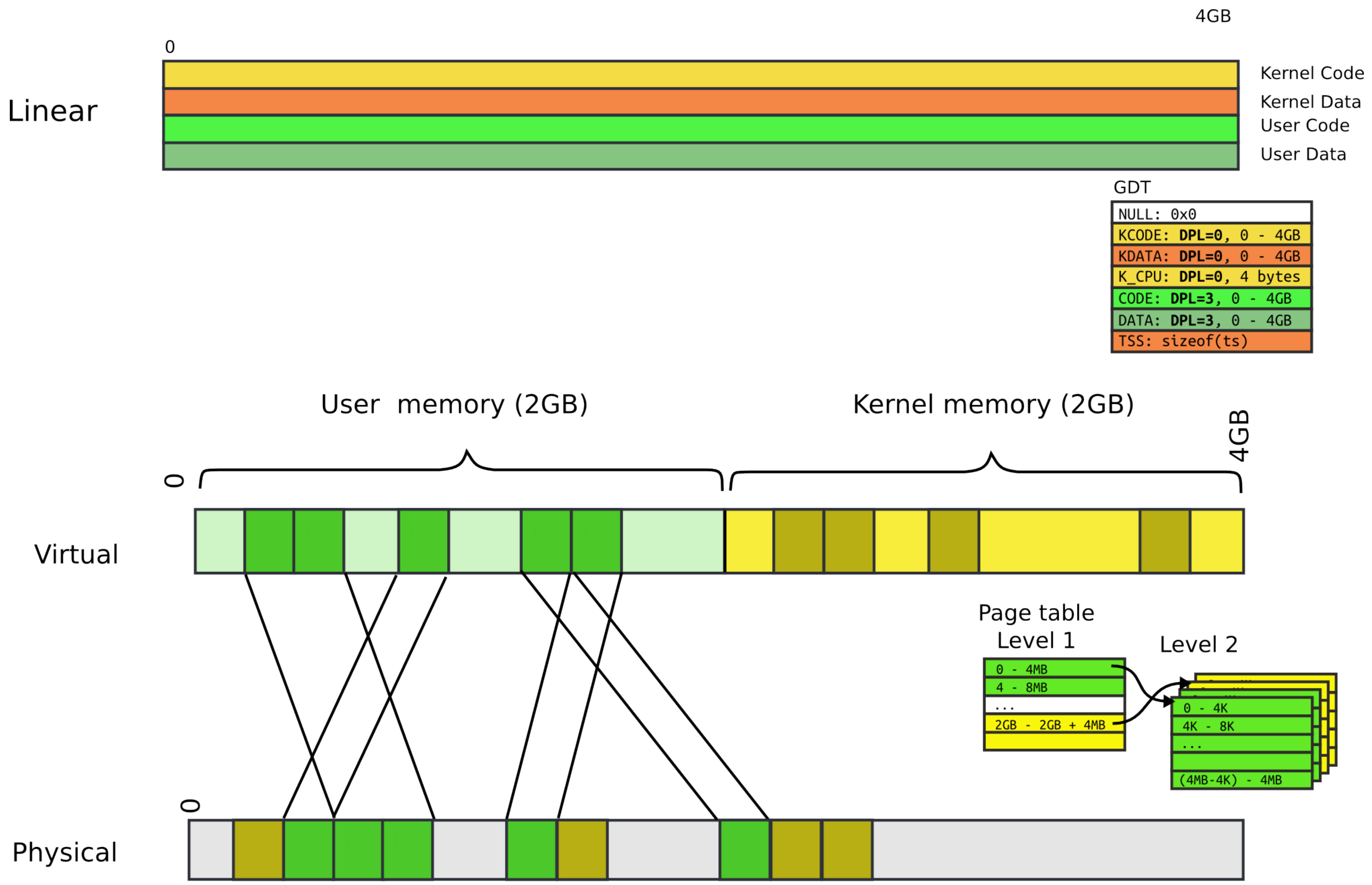
- Currently running code also has a privilege level
  - “Current privilege level” (CPL): 0-3
  - It is saved in the %cs register

Privilege level transitions

- CPL can access only less privileged segments
    - E.g., 0 can access 1, 2, 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

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
- **But then... how the kernel is protected?**

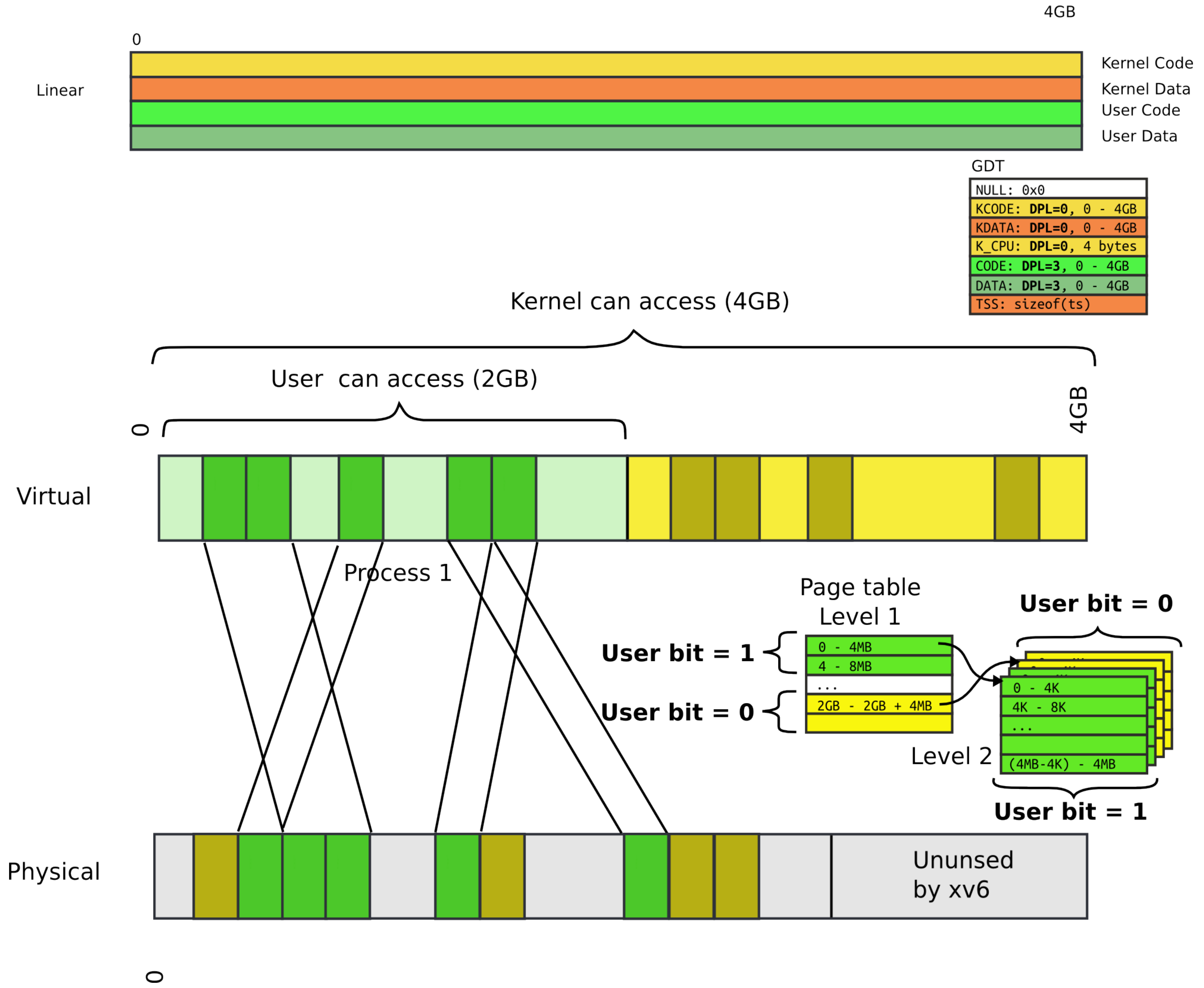


GDT

```
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)
```

Page table: user bit

- Each entry (both Level 1 and Level 2) has a bit
  - If set, code at privilege level 3 can access
  - If not, only levels 0-2 can access
- Note, only 2 levels, not 4 like with segments
- All kernel code is mapped with the user bit clear
  - This protects user-level code from accessing the kernel

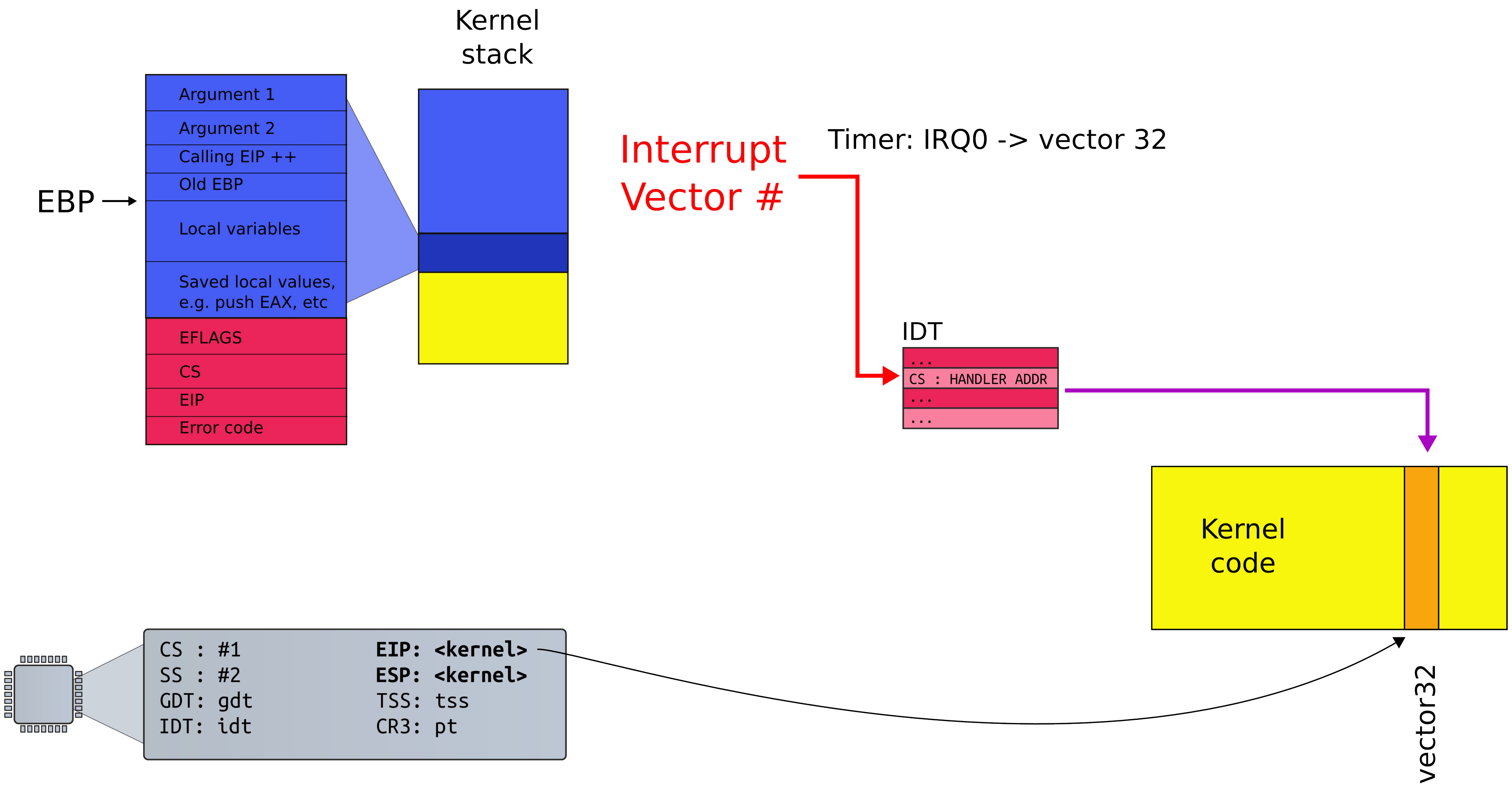


GDT

```
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)
```

Back to interrupts

Recap: interrupt path, no PL change

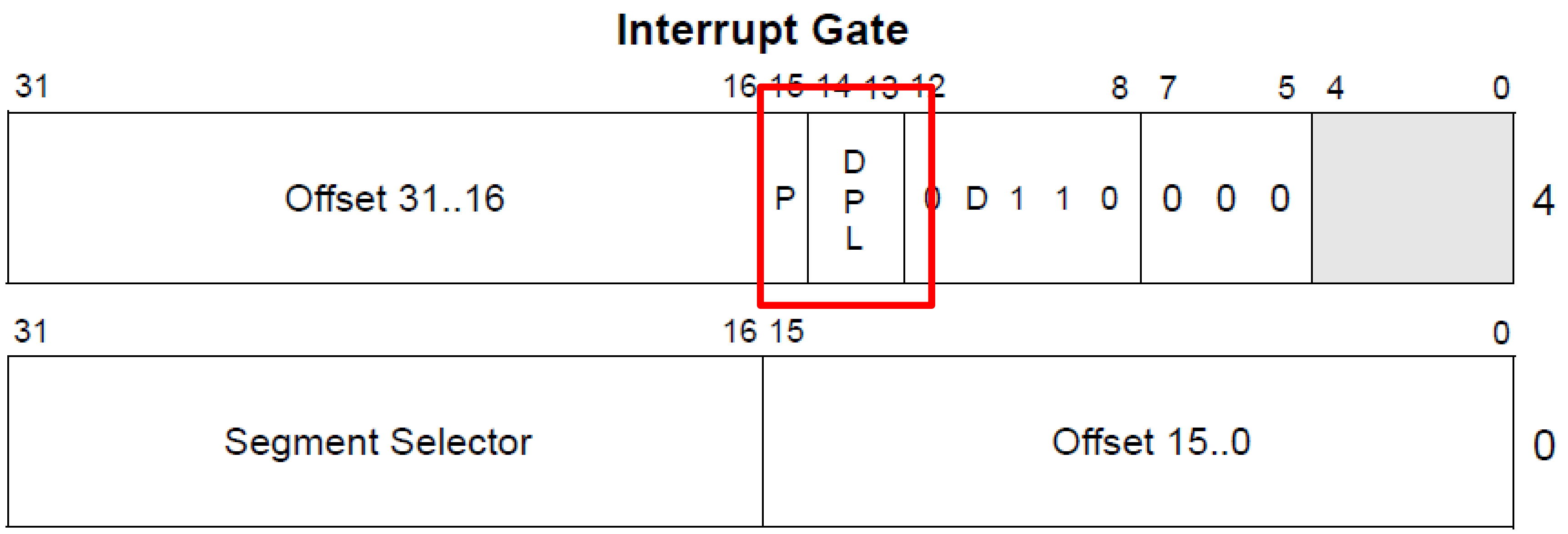


```
CS : #1          EIP: <kernel>
SS : #2          ESP: <kernel>
GDT: gdt         TSS: tss
IDT: idt         CR3: pt
```

Processing of interrupt (cross PL)

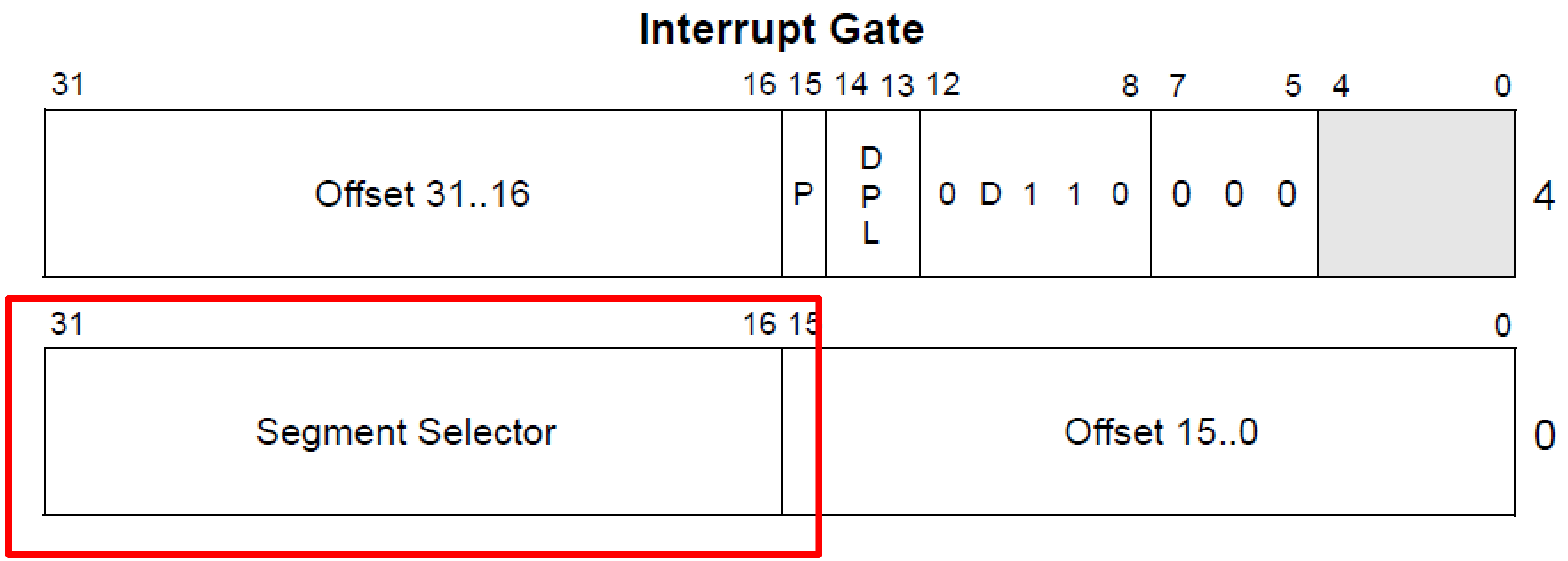
- Assume we're at CPL =3 (user)

Interrupt descriptor



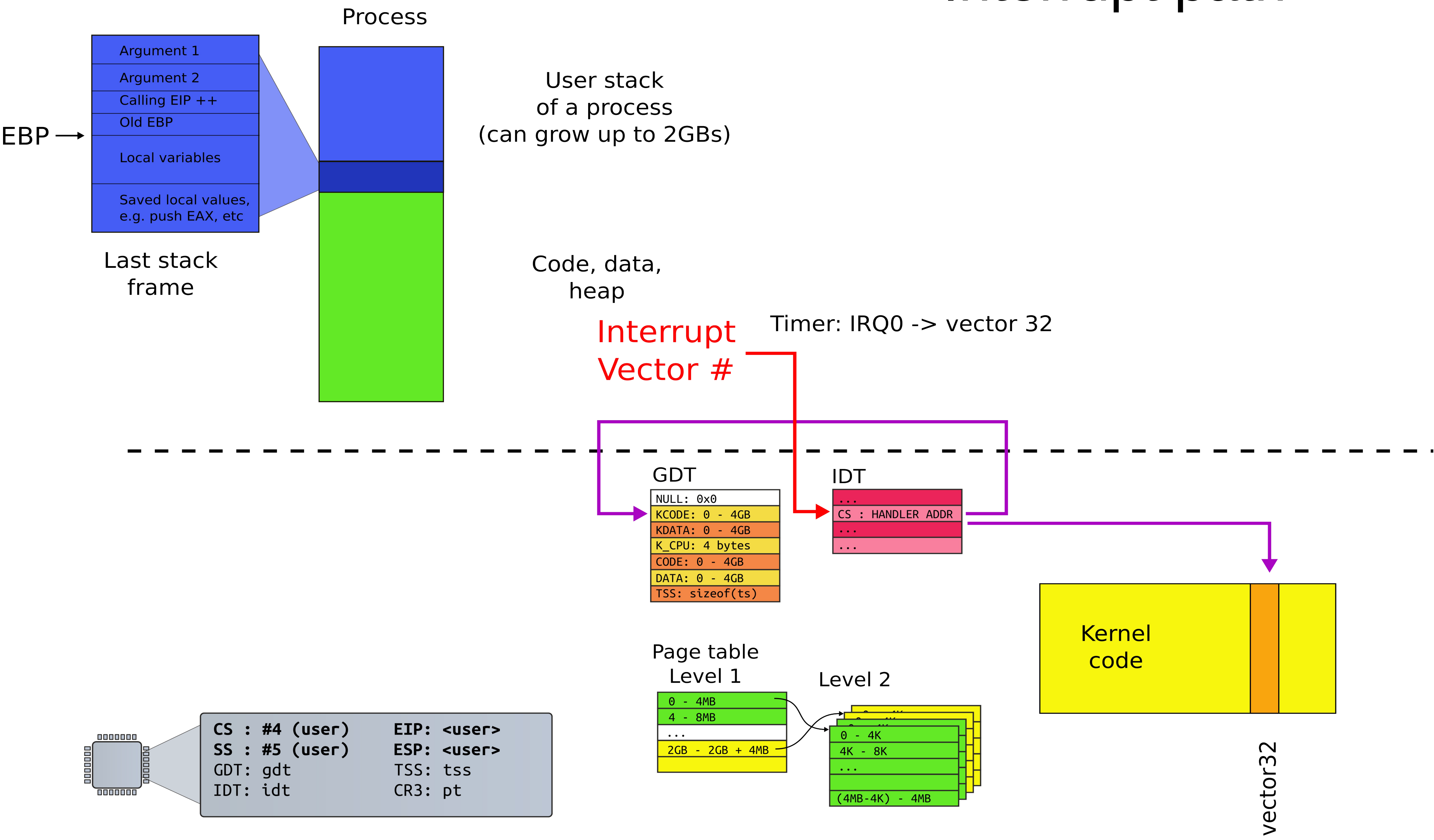
- Interrupt is allowed
  - If current privilege level (CPL) is less or equal to descriptor privilege level (DPL)
  - The kernel protects device interrupts from user

Interrupt descriptor



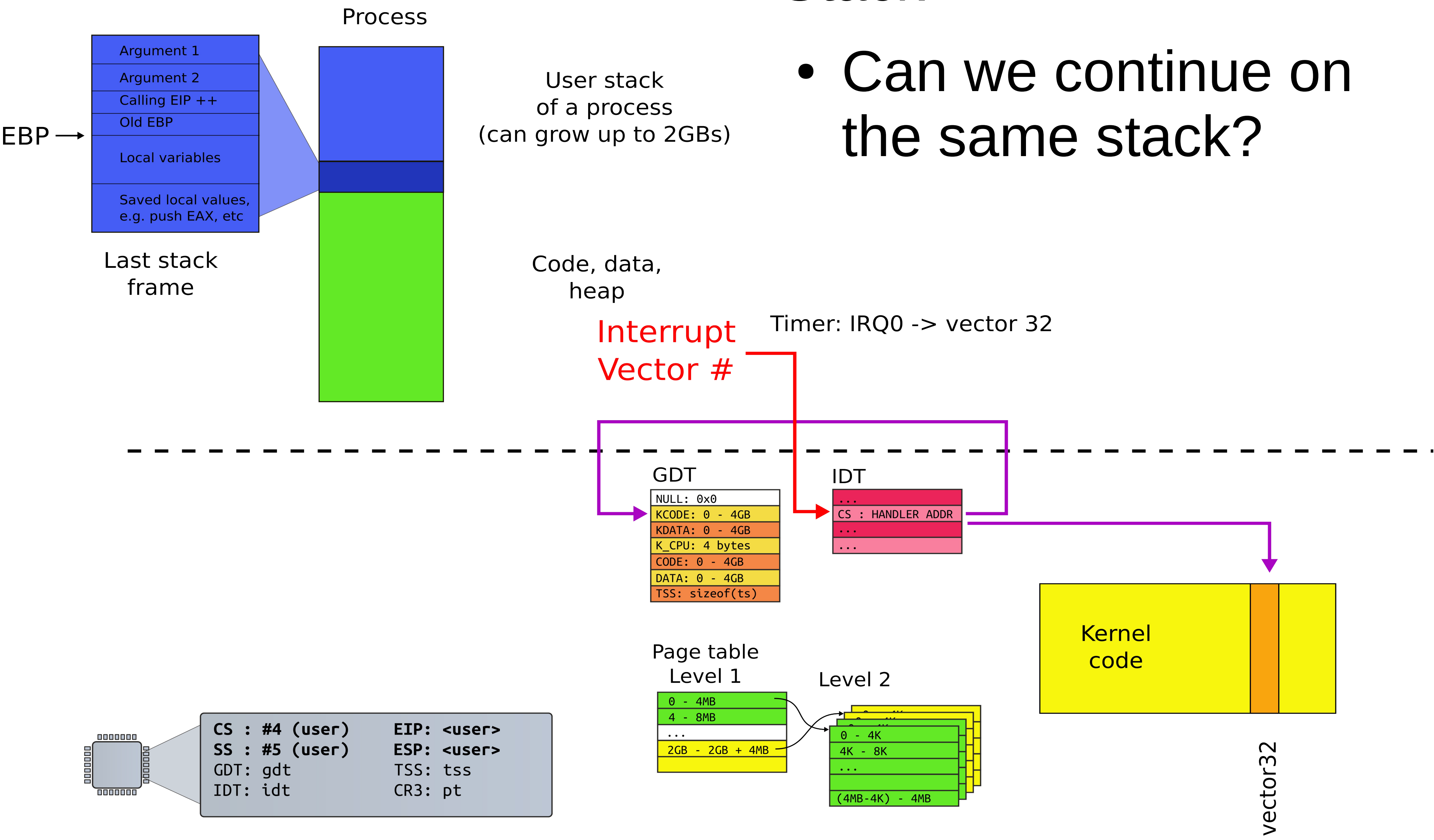
- Note that this new segment can be more privileged
  - E.g., CPL = 3, DPL = 3, new segment can be PL = 0
  - This is how user-code (PL=3) transitions into kernel (PL=0)

Interrupt path



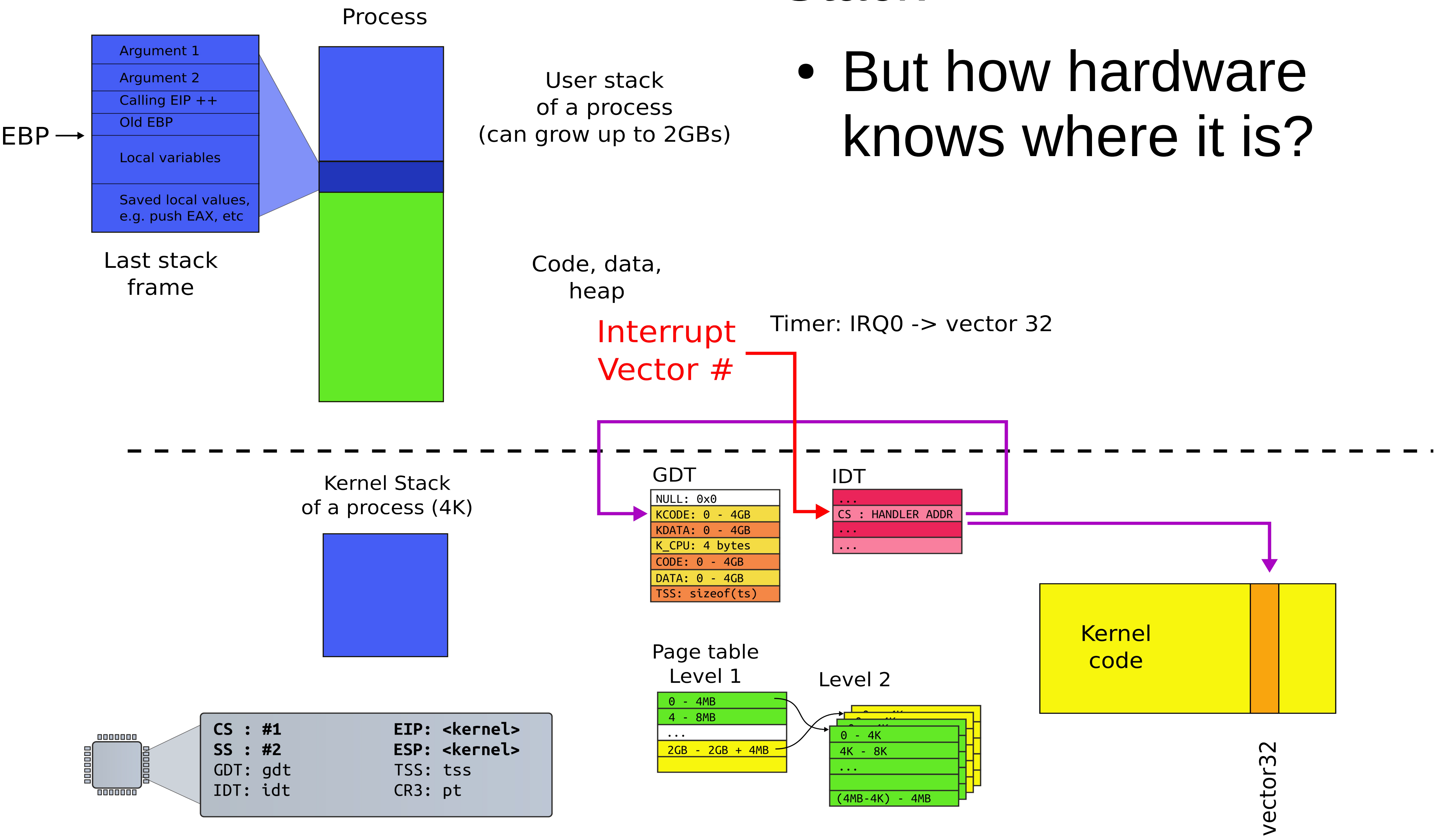
Stack

- Can we continue on the same stack?

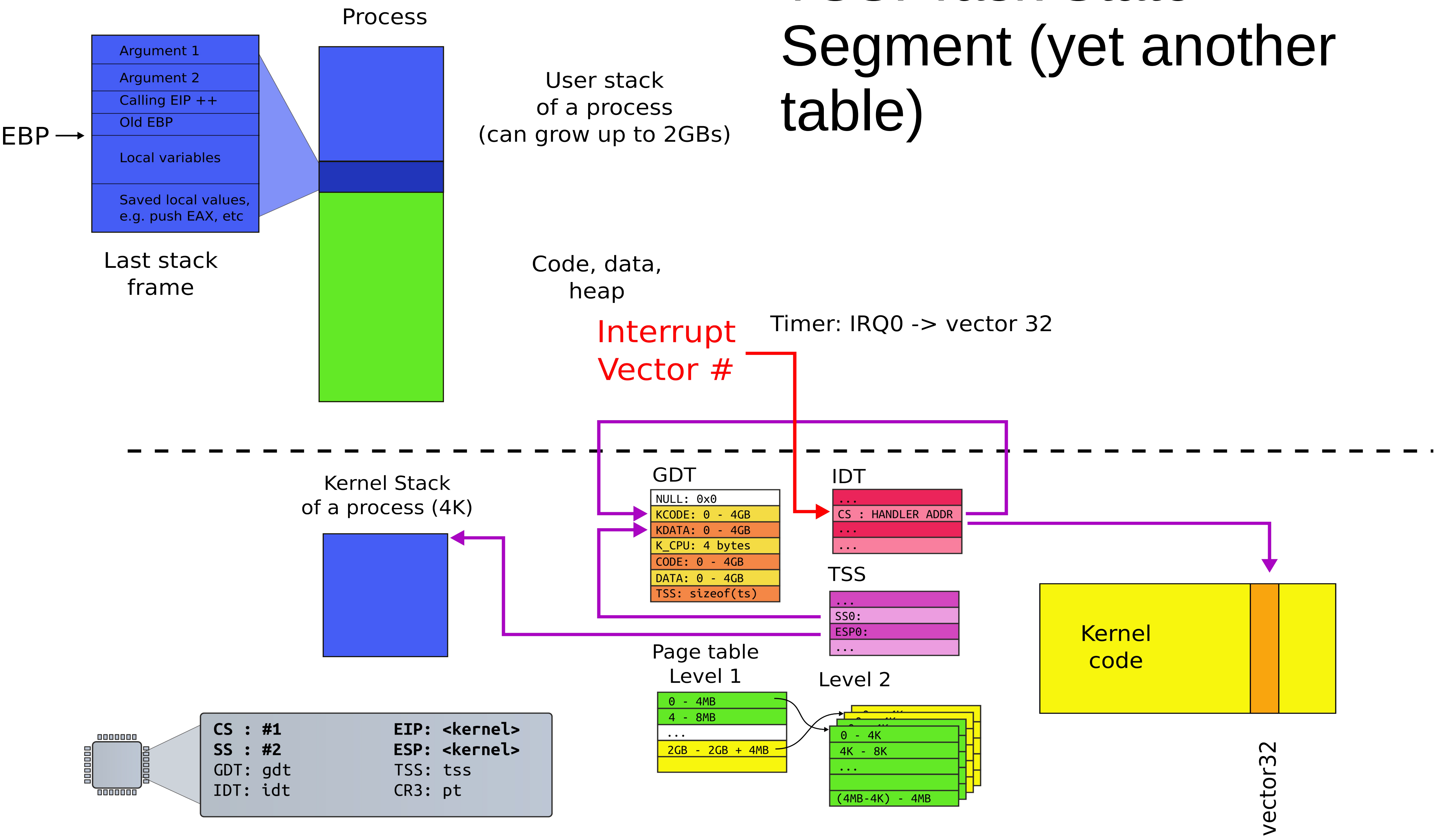


Stack

- But how hardware knows where it is?



TSS: Task State Segment (yet another table)



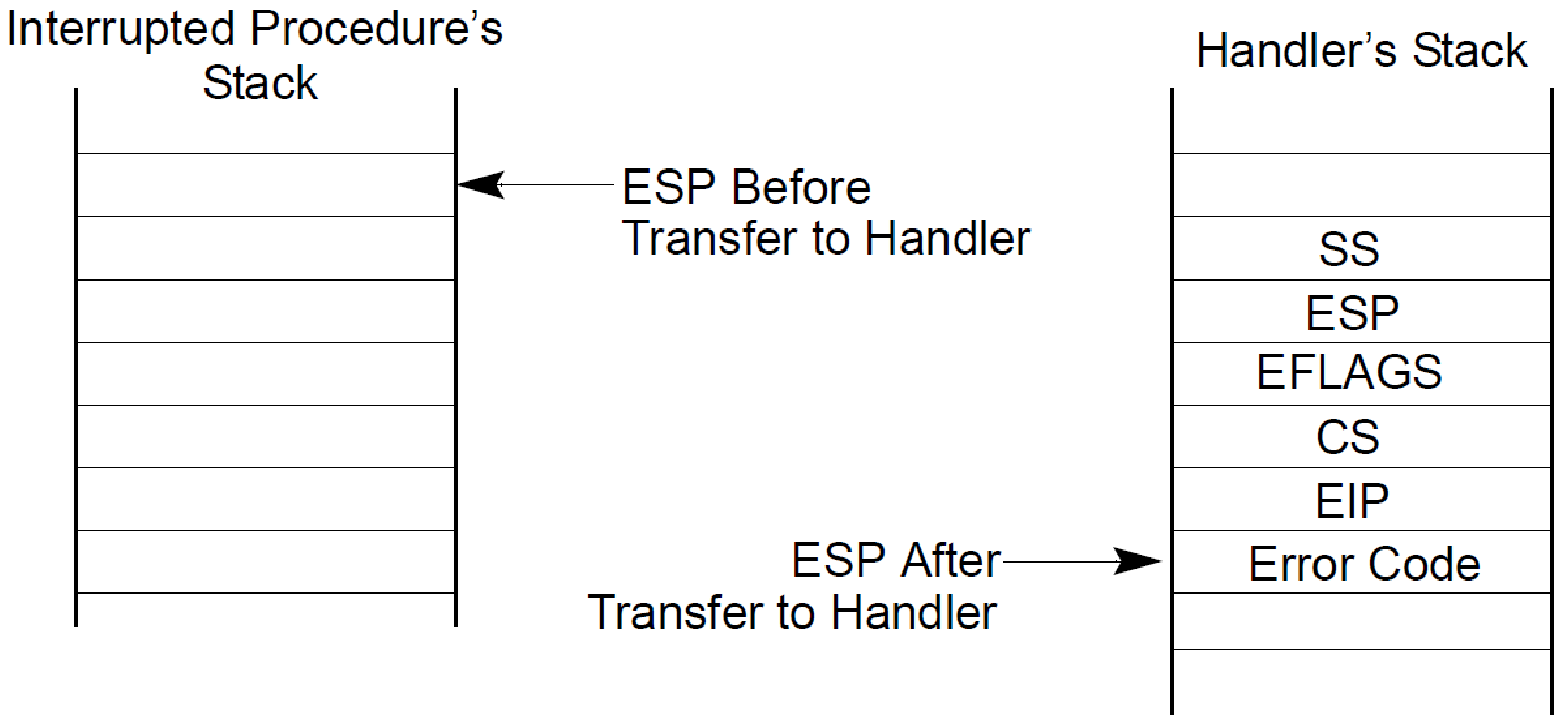
Task State Segment

- Another magic control block
  - Pointed to by special task register (TR)
- Lots of fields for rarely-used features
- A feature we care about in a modern OS:
  - Location of kernel stack (fields SS/ESP)
    - Stack segment selector
    - Location of the stack in that segment

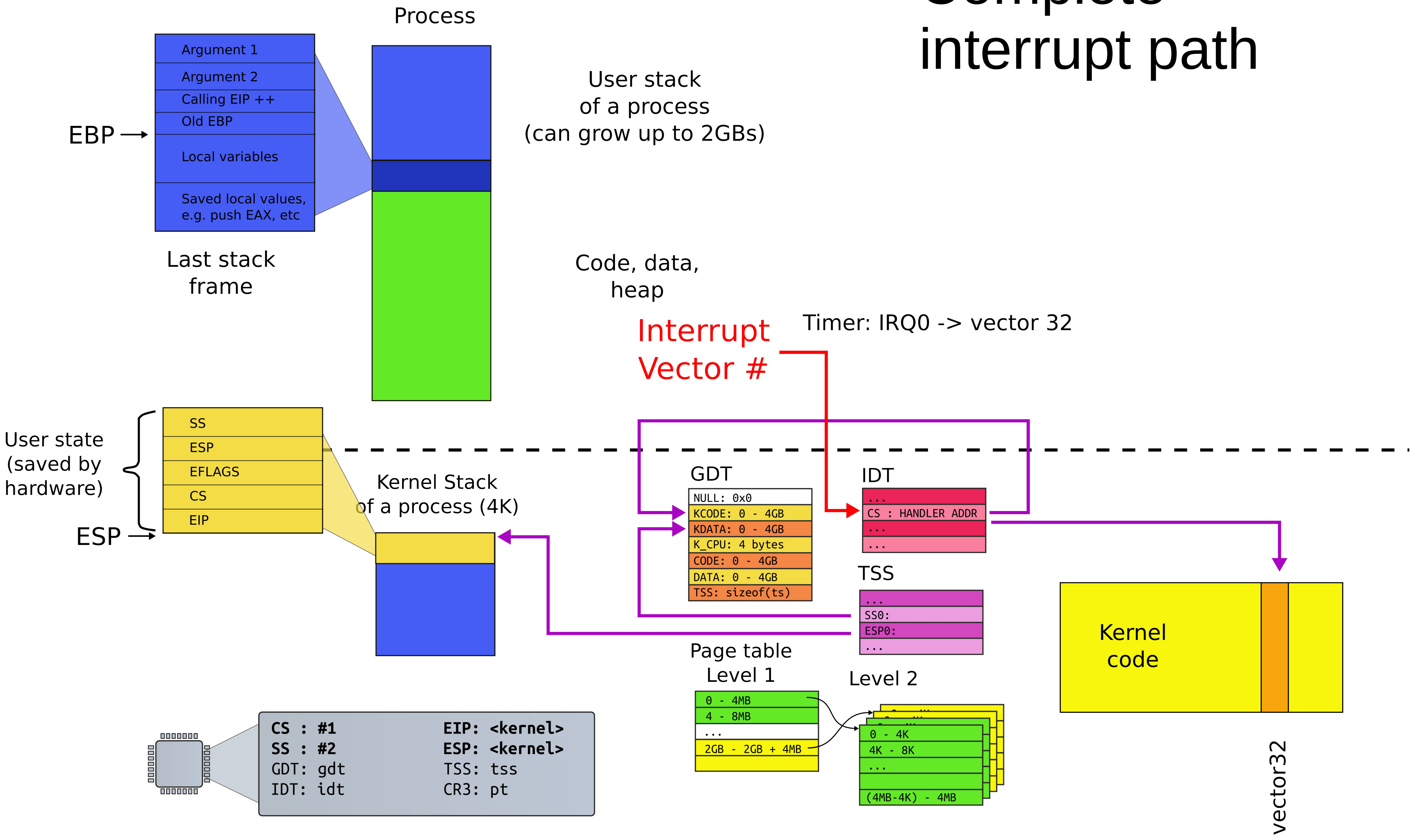
Processing of interrupt (cross PL)

1. Save ESP and SS in a CPU-internal register
2. Load SS and ESP from TSS
3. Push user SS, user ESP, user EFLAGS, user CS, user EIP onto new stack (kernel stack)
4. Set CS and EIP from IDT descriptor's segment selector and offset
5. If the call is through an interrupt gate clear some EFLAGS bits
6. Begin execution of a handler

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

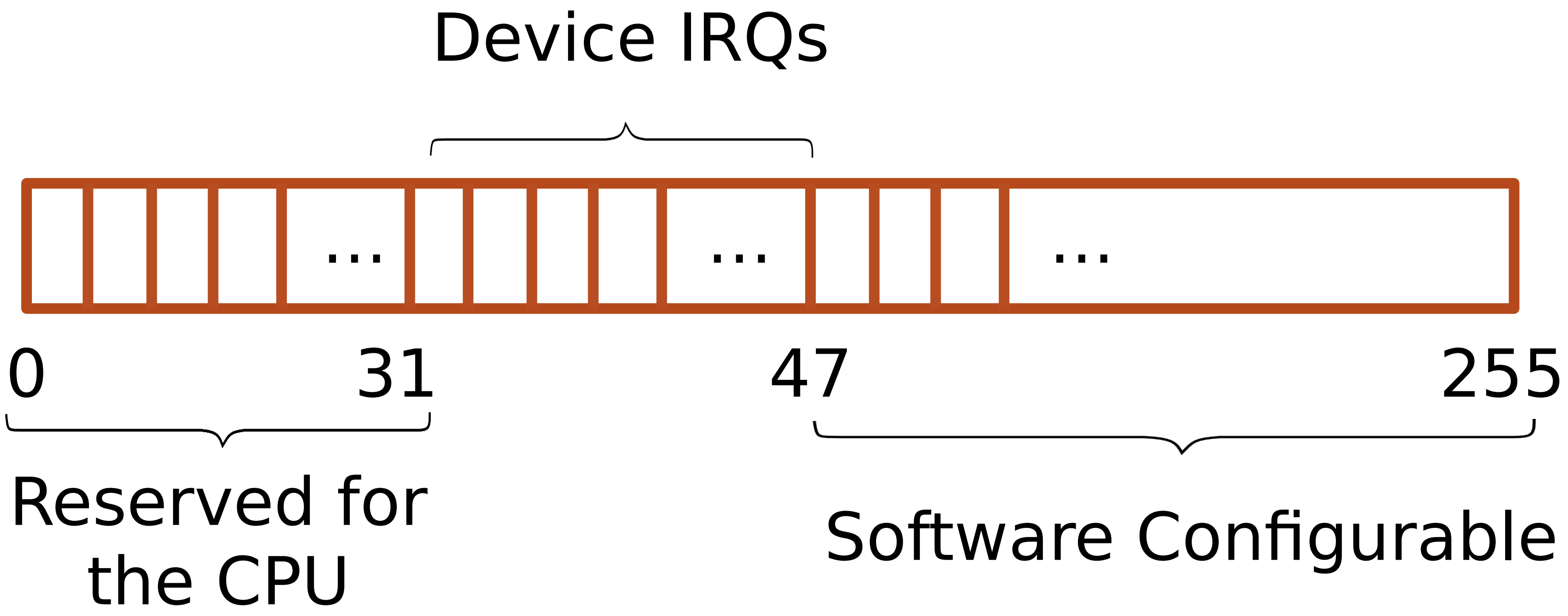
Complete interrupt path



Return from an interrupt

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

x86 interrupt table



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 for device interrupts (IRQs)
  - Most device’s IRQ line can be configured
  - Look up APICs for more info (Ch 4 of Bovet and Cesati)
  - 0x80 issues system call in Linux (more on this later)

Sources

- Interrupts
  - External
    - From a device
    - Through CPU pins connected to APIC
  - Software generated with INT n instruction
- Exceptions
  - Processor generated, when CPU detects an error in the program
    - Fault, trap, abort
  - Software generated with INTO, INT 3, BOUND

Software interrupts

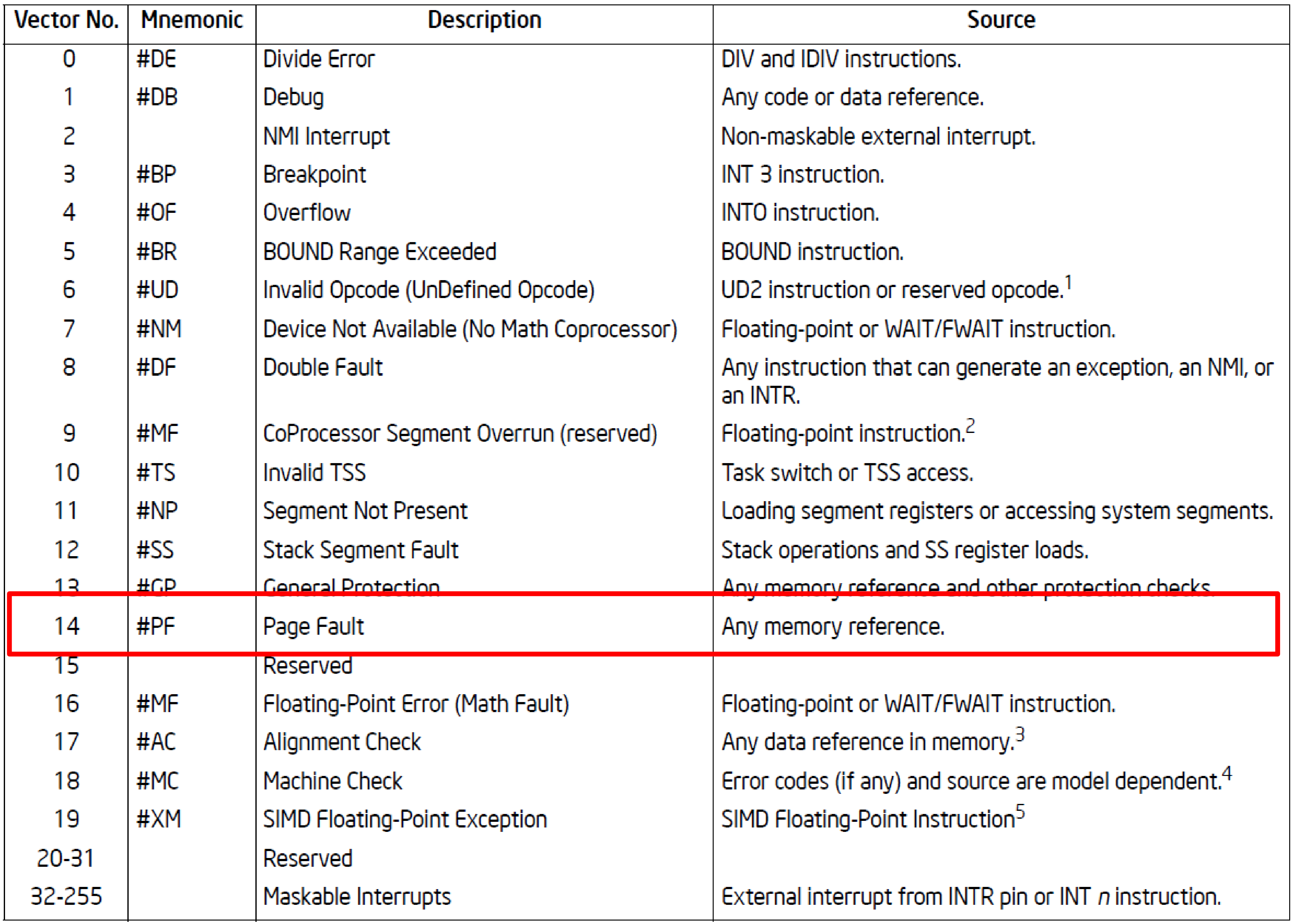
- The INT n instruction allows software to raise an interrupt
  - 0x80 is just a Linux convention
  - You could change it to use 0x81!
- There are a lot of spare indexes

- OS sets ring level required to raise an interrupt
  - Generally, user programs can’t issue an int 14 (page fault manually)
  - An unauthorized int instruction causes a general protection fault
    - Interrupt 13

Disabling interrupts

- Delivery of maskable interrupts can be disabled with IF (interrupt flag) in EFLAGS register
- Exceptions
  - Non-maskable interrupts (see next slide)
  - INT n – cannot be masked as it is synchronous

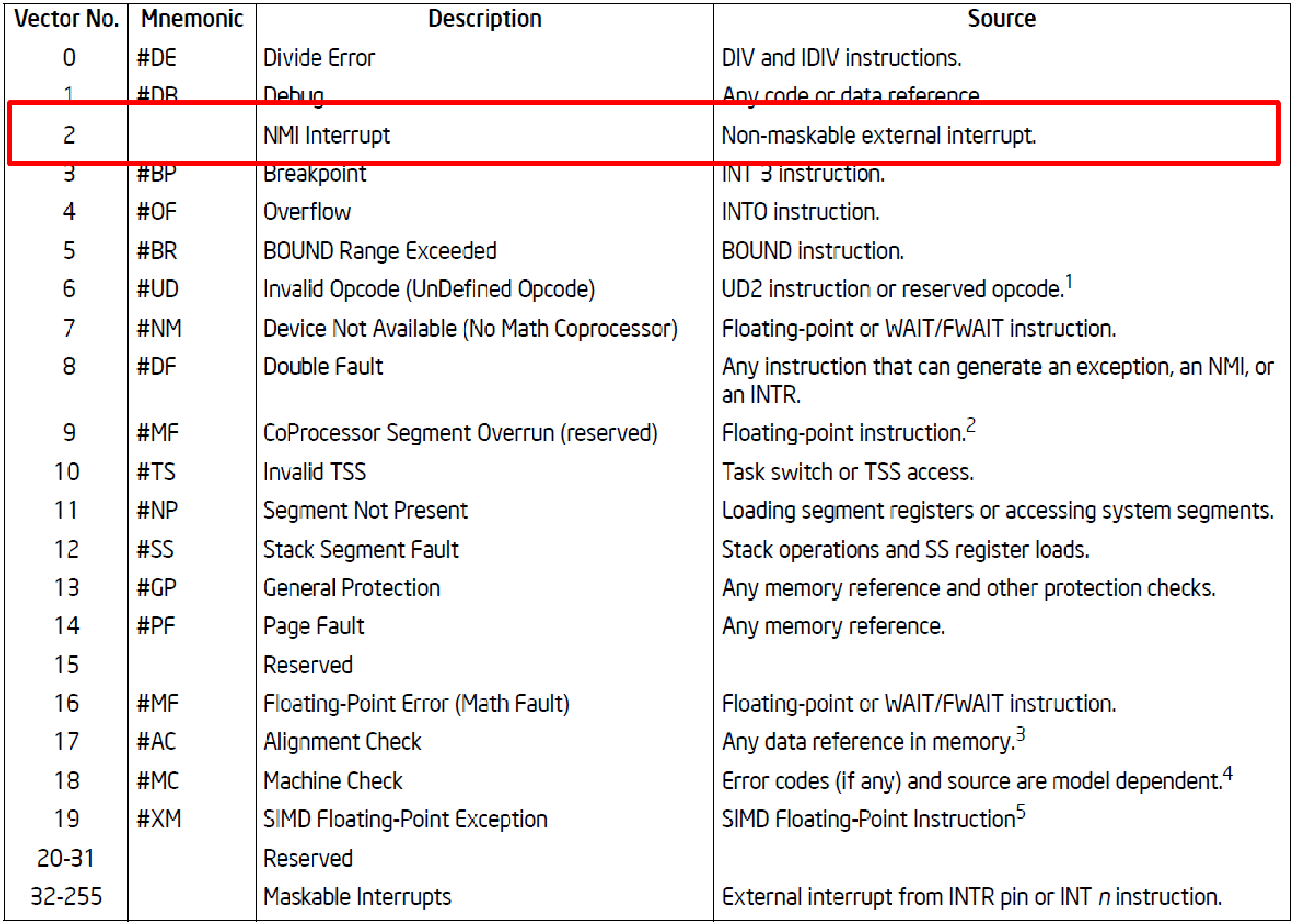
Vector No.	Mnemonic	Description	Source
0	#DE	Divide Error	DIV and IDIV instructions.
1	#DB	Debug	Any code or data reference.
2		NMI Interrupt	Non-maskable external interrupt.
3	#BP	Breakpoint	INT 3 instruction.
4	#OF	Overflow	INTO instruction.
5	#BR	BOUND Range Exceeded	BOUND instruction.
6	#UD	Invalid 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.

Nonmaskable interrupts (NMI)

- Delivered even if IF is clear, e.g. interrupts disabled
  - CPU blocks subsequent NMI interrupts until IRET
- Sources
  - External hardware asserts the NMI pin
  - Processor receives a message on the system bus, or the APIC serial bus with NMI delivery mode
- Delivered via vector #2



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.

Thank you.