

# cs5460/6460: Operating Systems

## Lecture: Interrupts and Exceptions

Anton Burtsev

March, 2024

```
1317 main(void)
1318 {
1319     kinit1(end, P2V(4*1024*1024)); // phys page allocator
1320     kvmalloc(); // kernel page table
1321     mpinit(); // detect other processors
1322     lapicinit(); // interrupt controller
1323     seginit(); // segment descriptors
1324     cprintf("\ncpu%d: starting xv6\n\n", cpunum()); // main()
1325     picinit(); // another interrupt controller
1326     ioapicinit(); // another interrupt controller
1327     consoleinit(); // console hardware
1328     uartinit(); // serial port
1329     pinit(); // process table
1330     tvinit(); // trap vectors
1331     binit(); // buffer cache
1332     fileinit(); // file table
1333     ideinit(); // disk
1334     if(!ismp)
1335         timerinit(); // uniprocessor timer
```

# Why do we need interrupts?

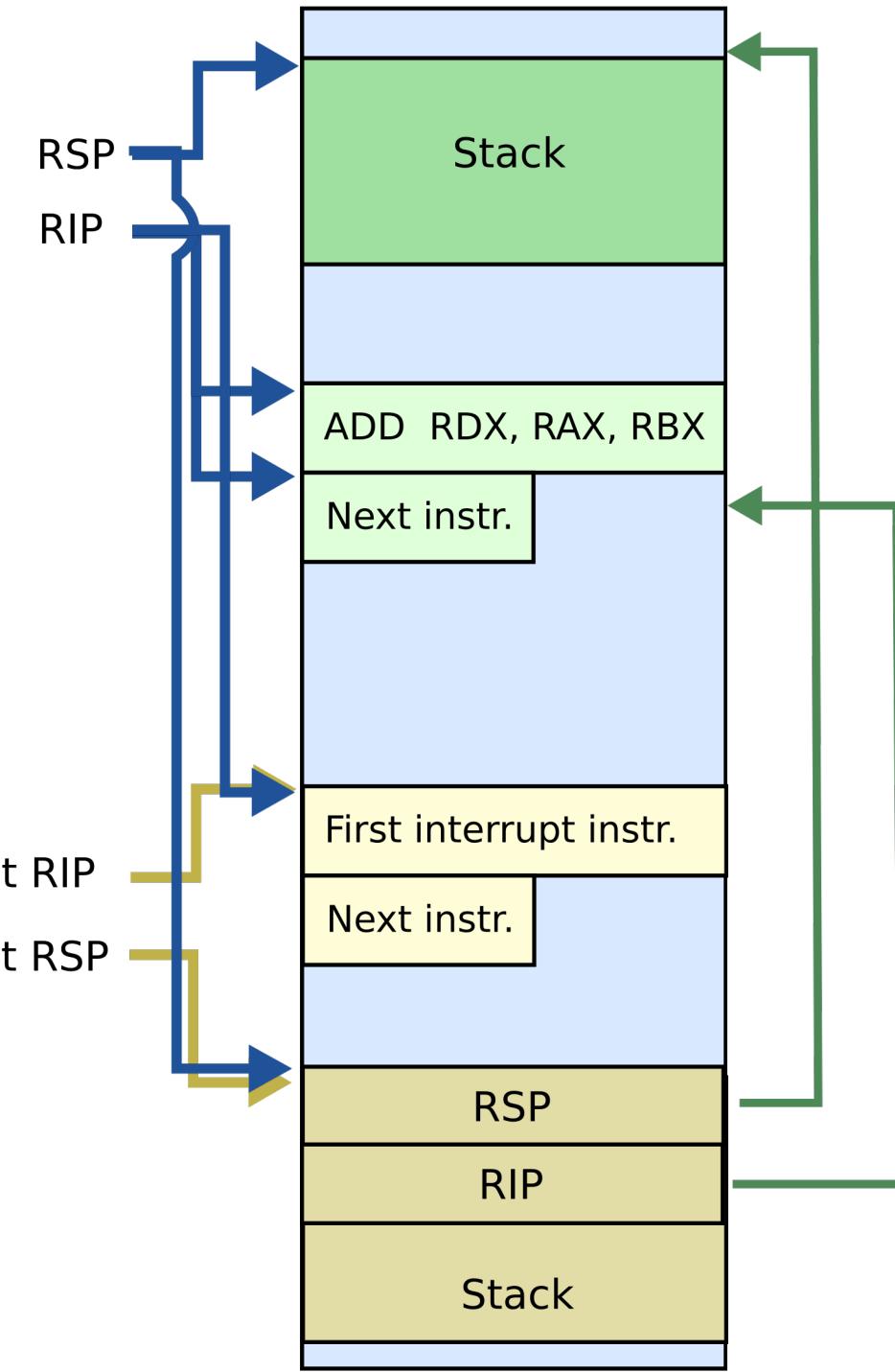
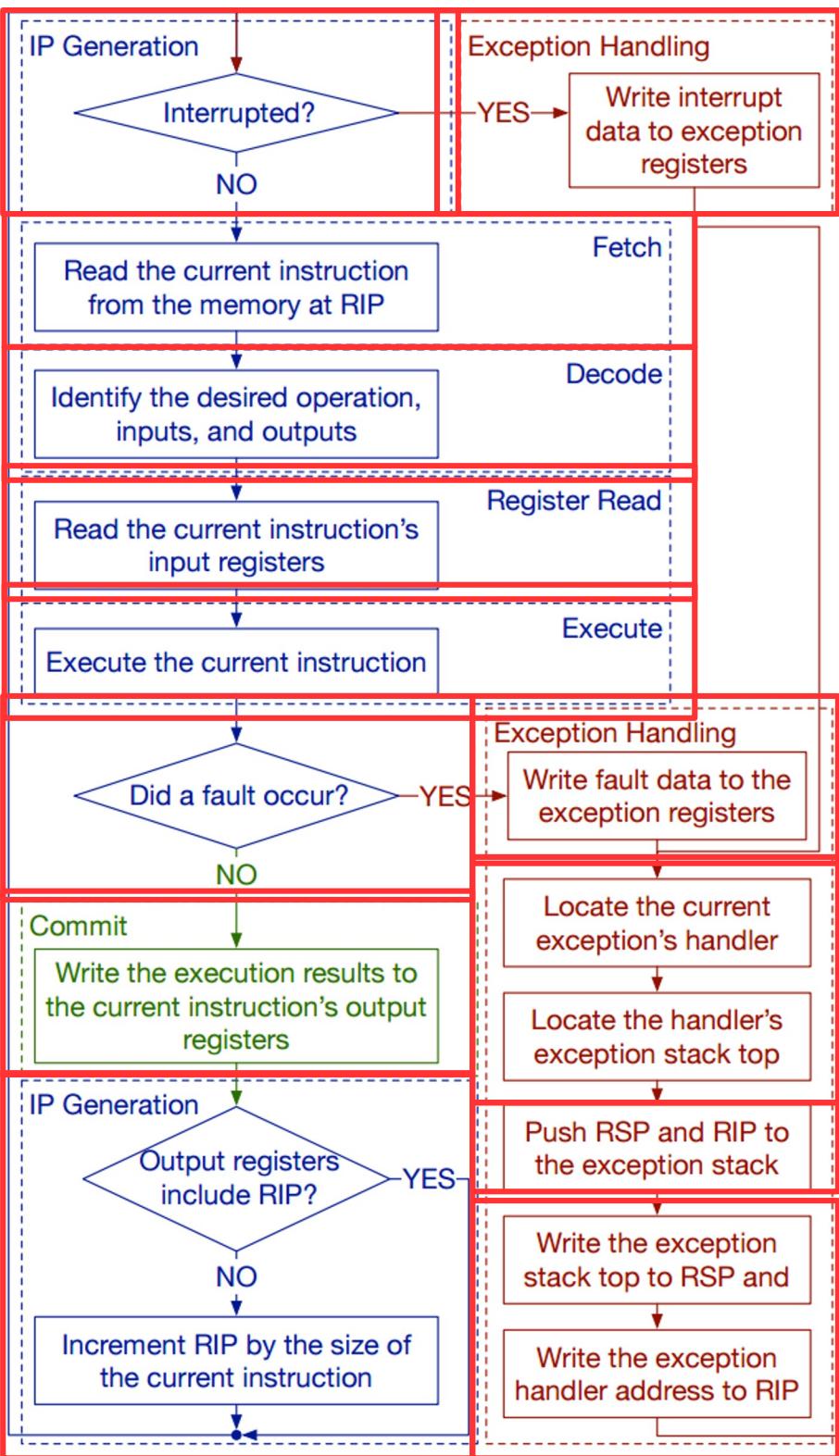
Remember:

hardware interface is designed to help OS

# 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 (OS maps the page if its on disk)
  - [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
  - In other words, to implement system calls
  - Now, a faster mechanism is available (sysenter)

How do we handle an interrupt?



# Handling interrupts and exceptions

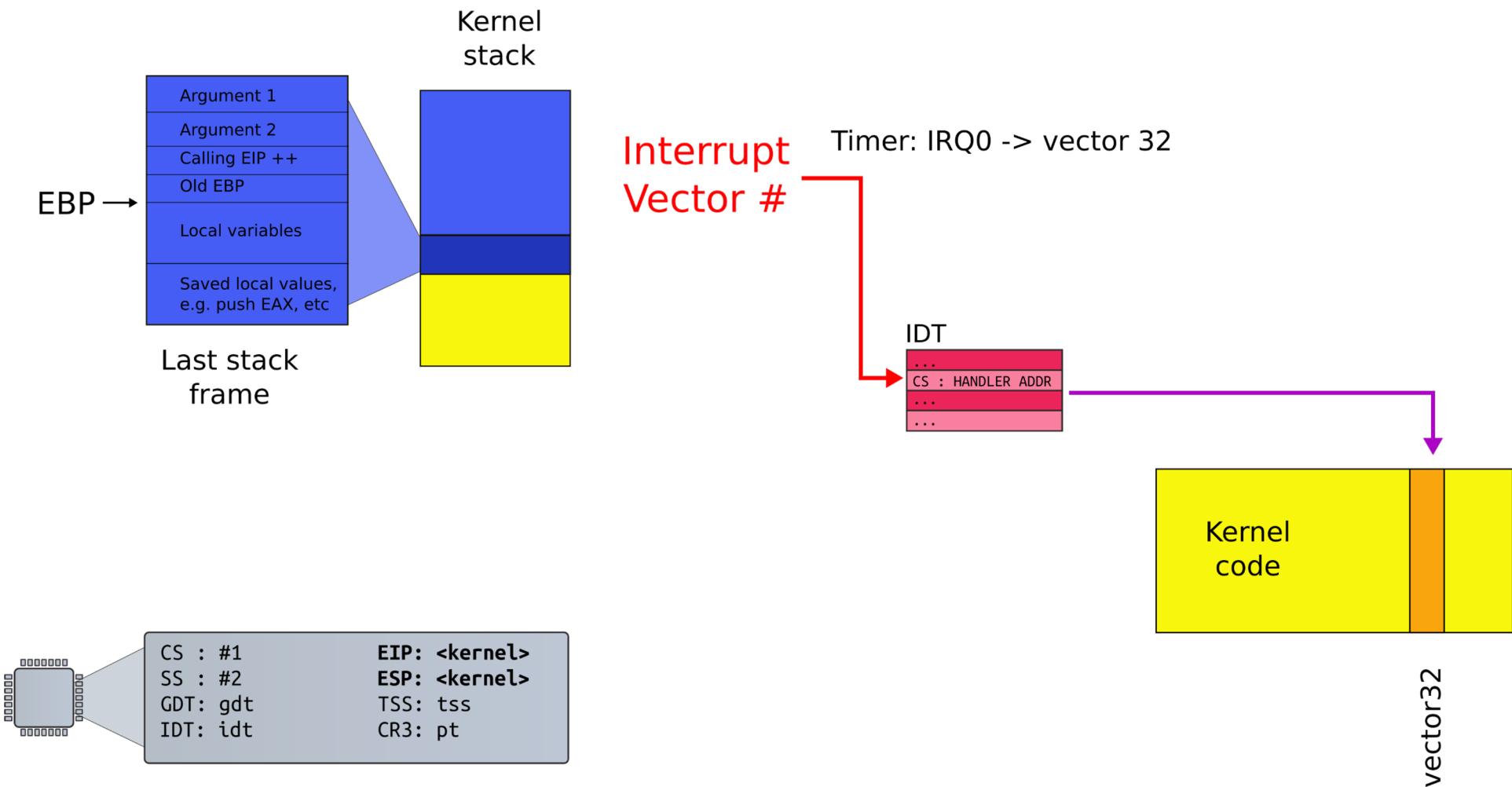
- In both synchronous and asynchronous cases the CPU follows the **same procedure**
  - Stop execution of the current program
  - Start execution of a handler
  - Processor accesses the handler through an entry in the Interrupt Descriptor Table (IDT)
    - Each interrupt is defined by a number
    -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

# Case #1: Interrupt path no change in privilege level

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



IDTR Register

47

16 15

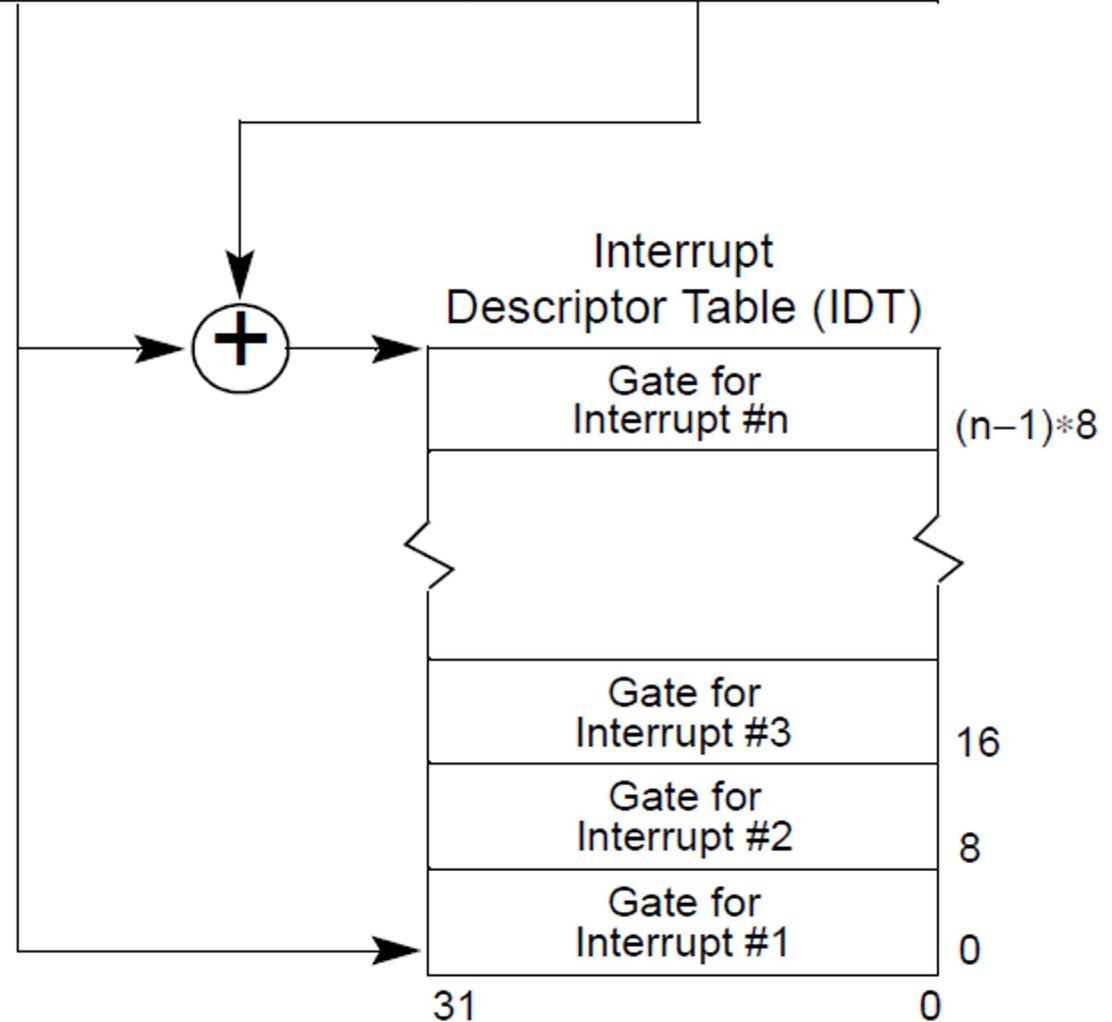
0

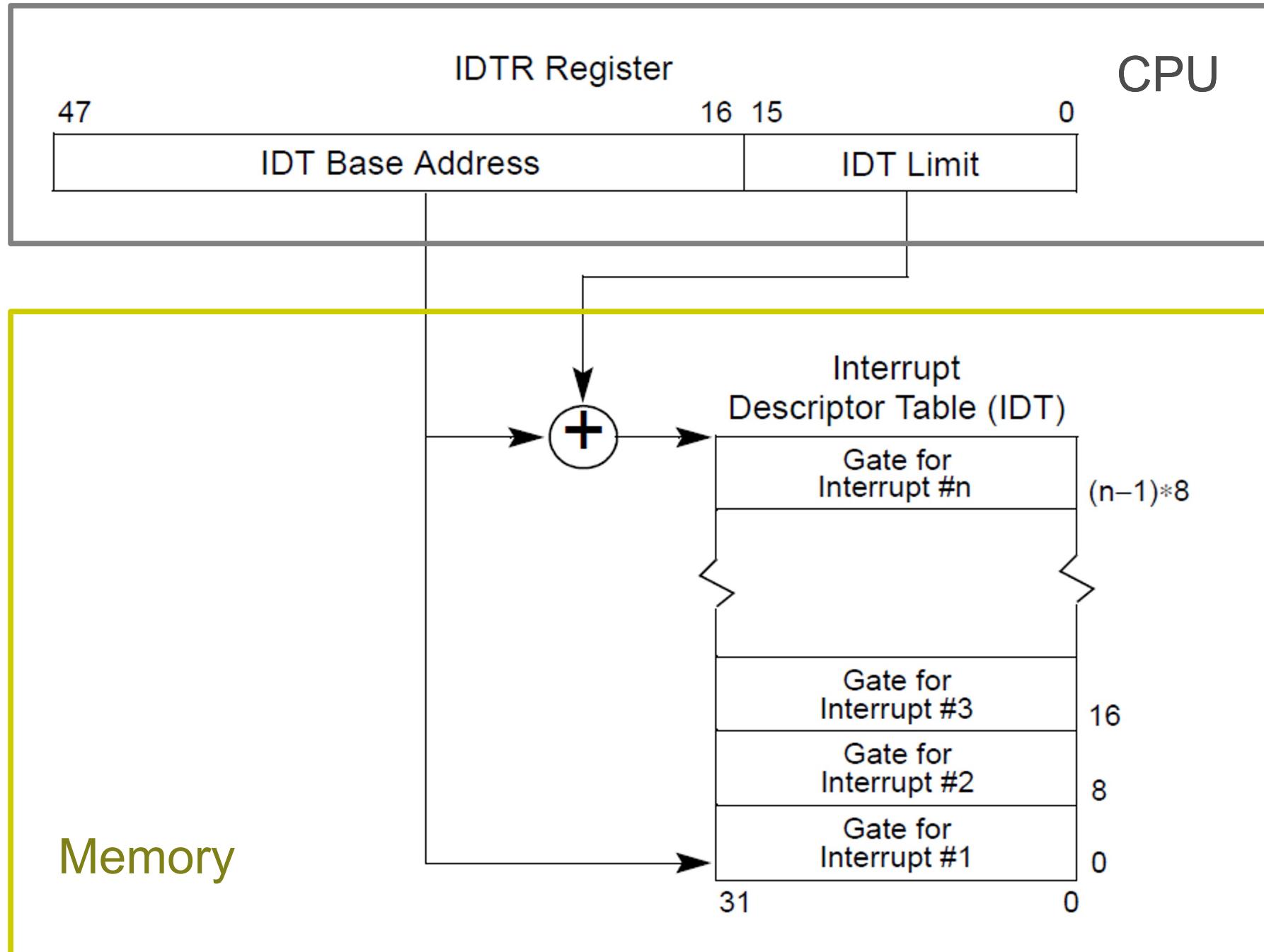
IDT Base Address

IDT Limit

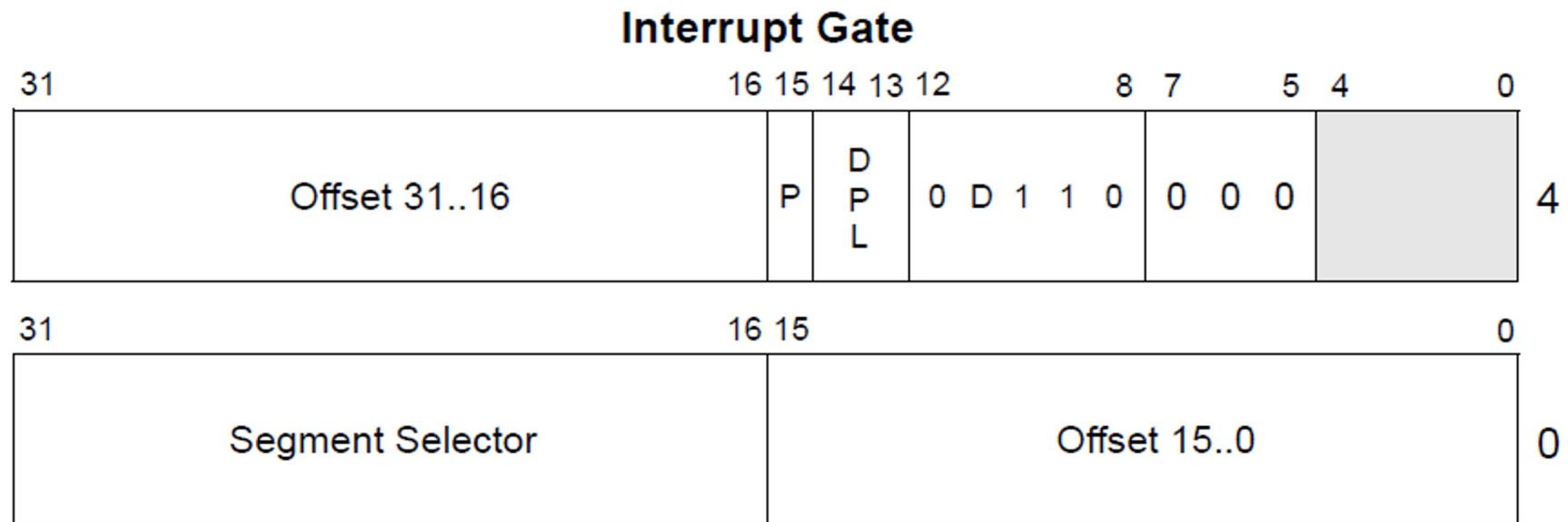
### Interrupt descriptor table (IDT)

- Is pointed by the IDTR register
- Virtual address
- OS configures the value and loads it into the register (normally during boot)

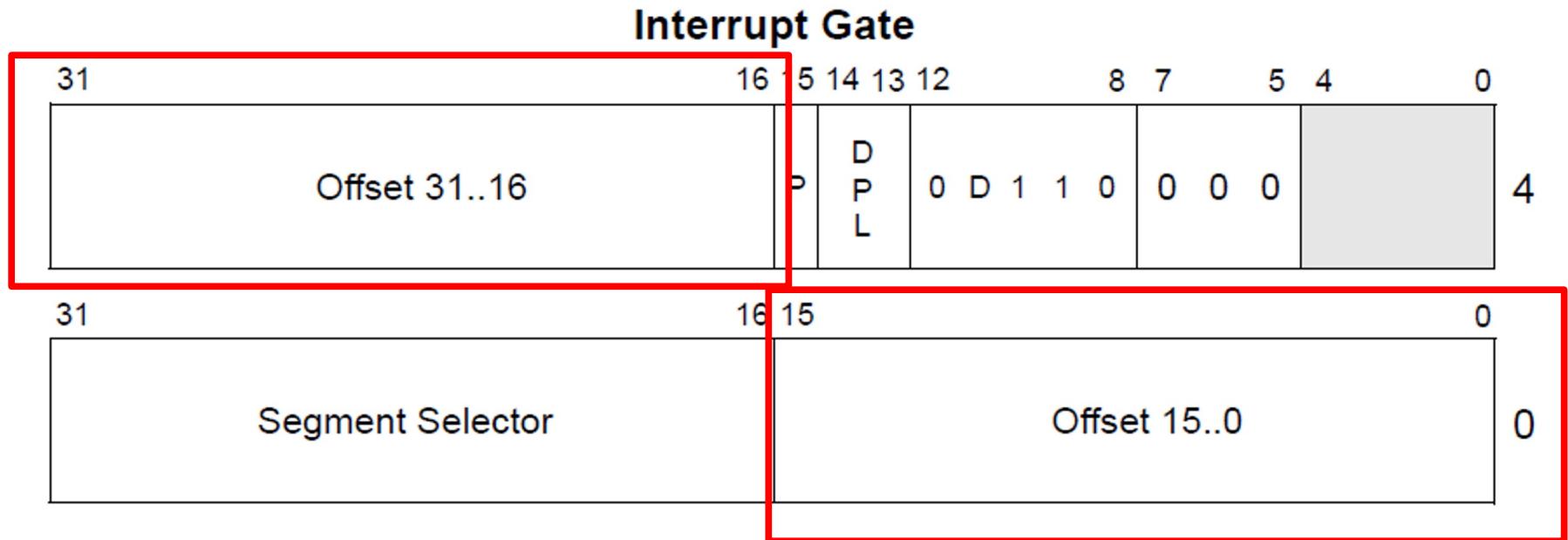




# Interrupt descriptor



# Interrupt descriptor

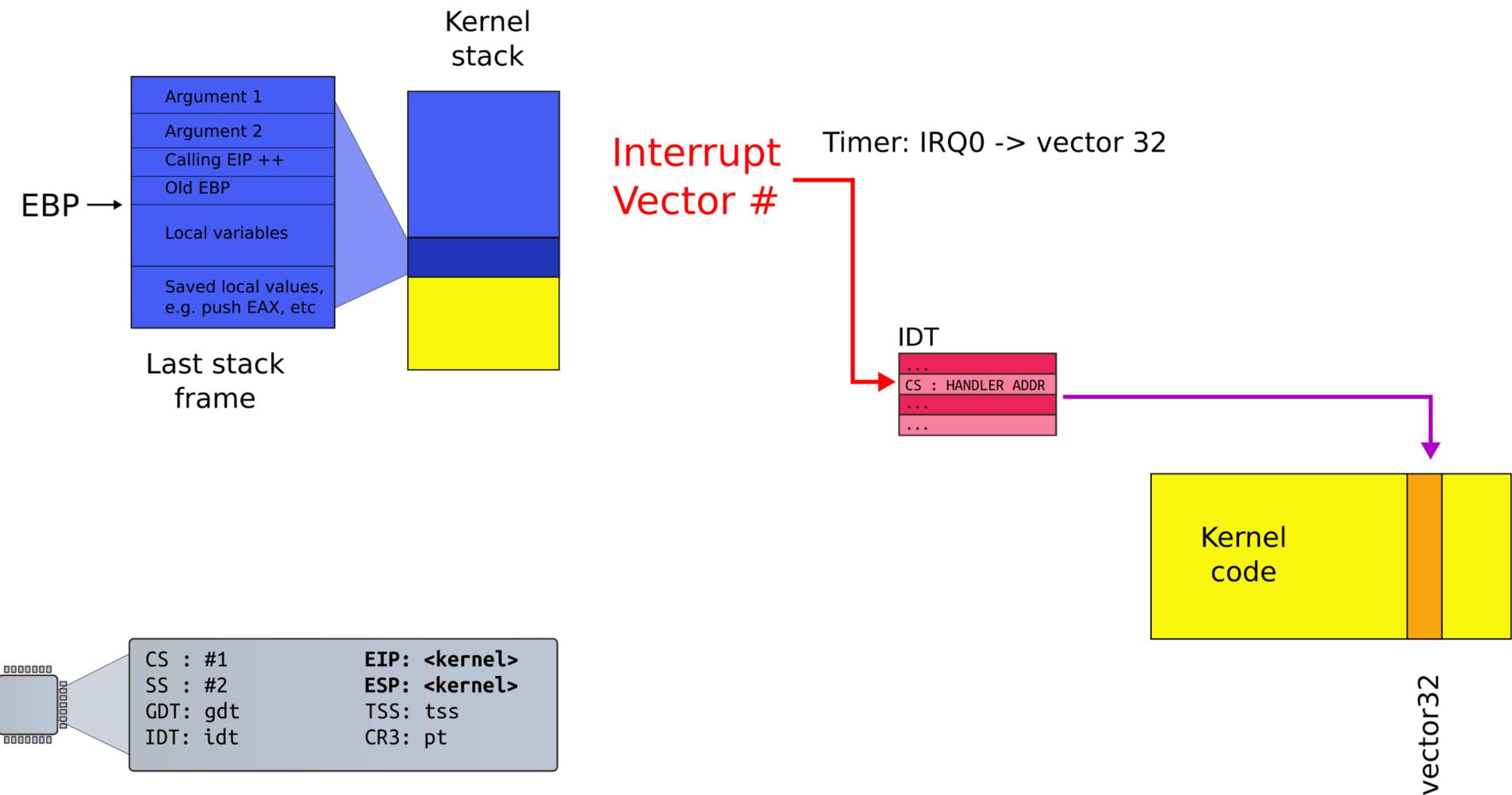


- We will walk through these fields gradually
- For now, we care about **vector offset**
- Pointer to the interrupt handler

# Interrupt handlers

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

# Interrupt path

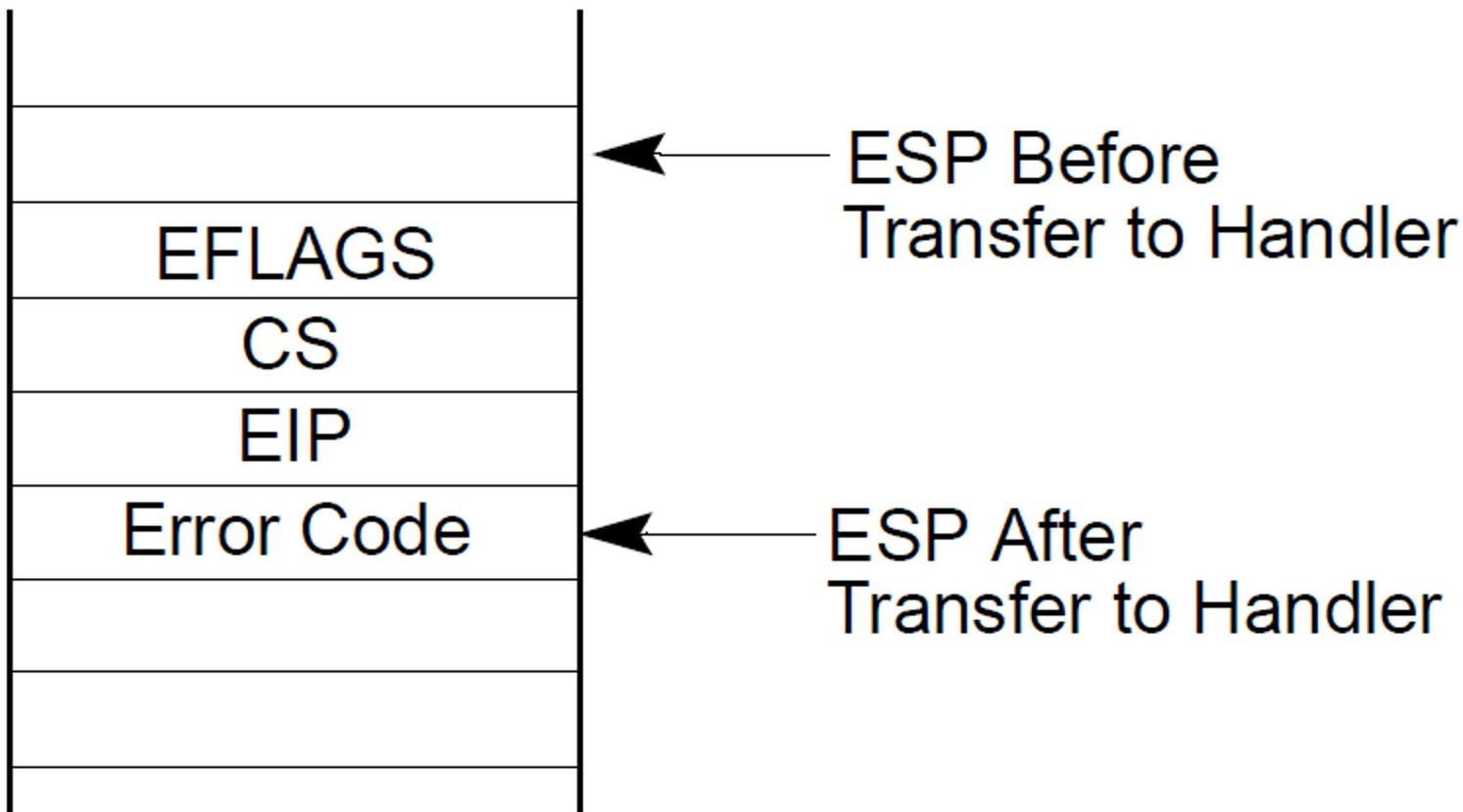


# Processing of interrupt (same PL)

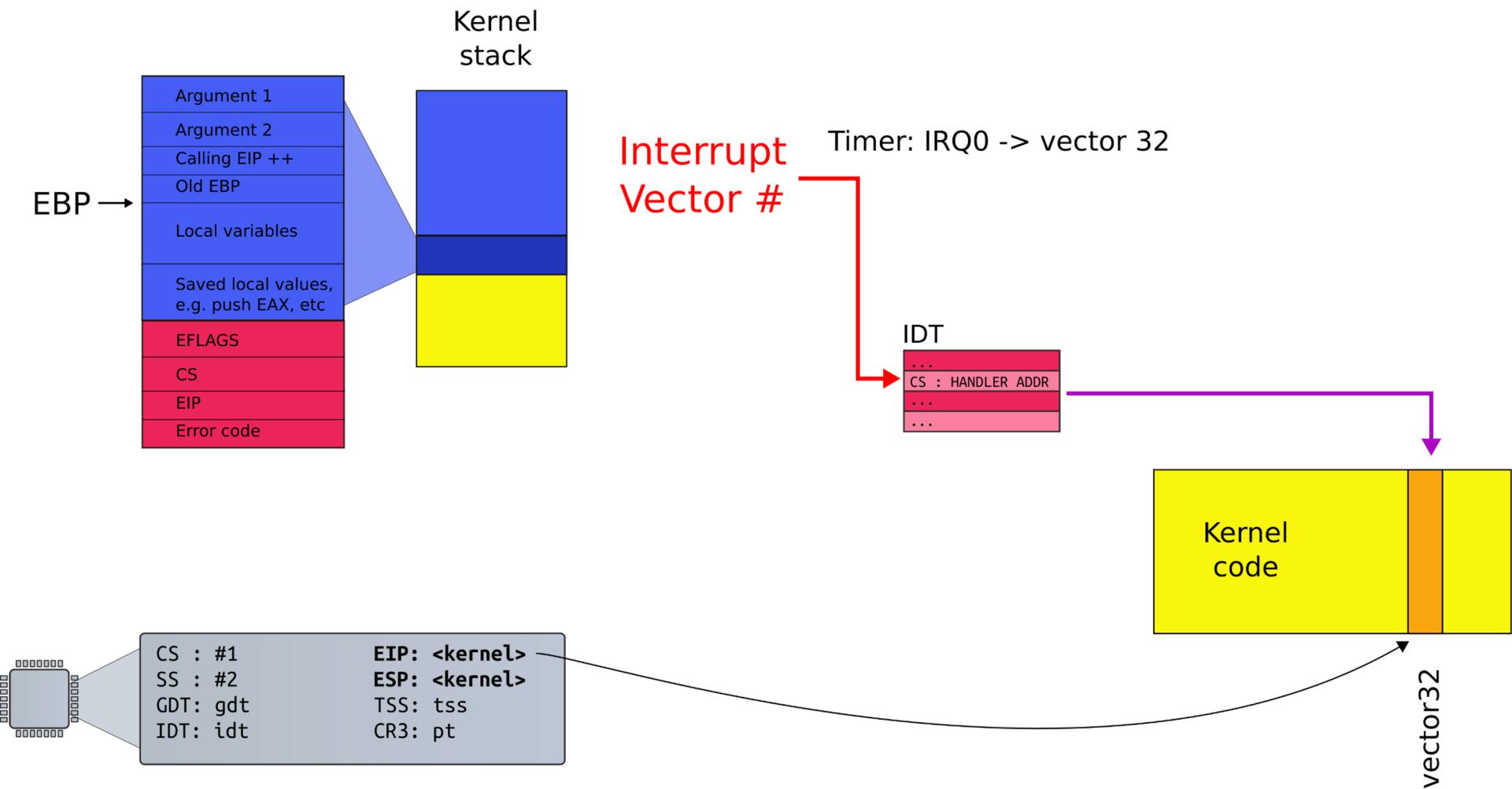
1. Push the current contents of the **EFLAGS**, **CS**, and **EIP** registers (in that order) on the stack
2. Push an **error code** (if appropriate) on the stack
3. Load the **segment selector** for the new code segment and the new **instruction pointer** (from the interrupt gate or trap gate) into 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



# 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

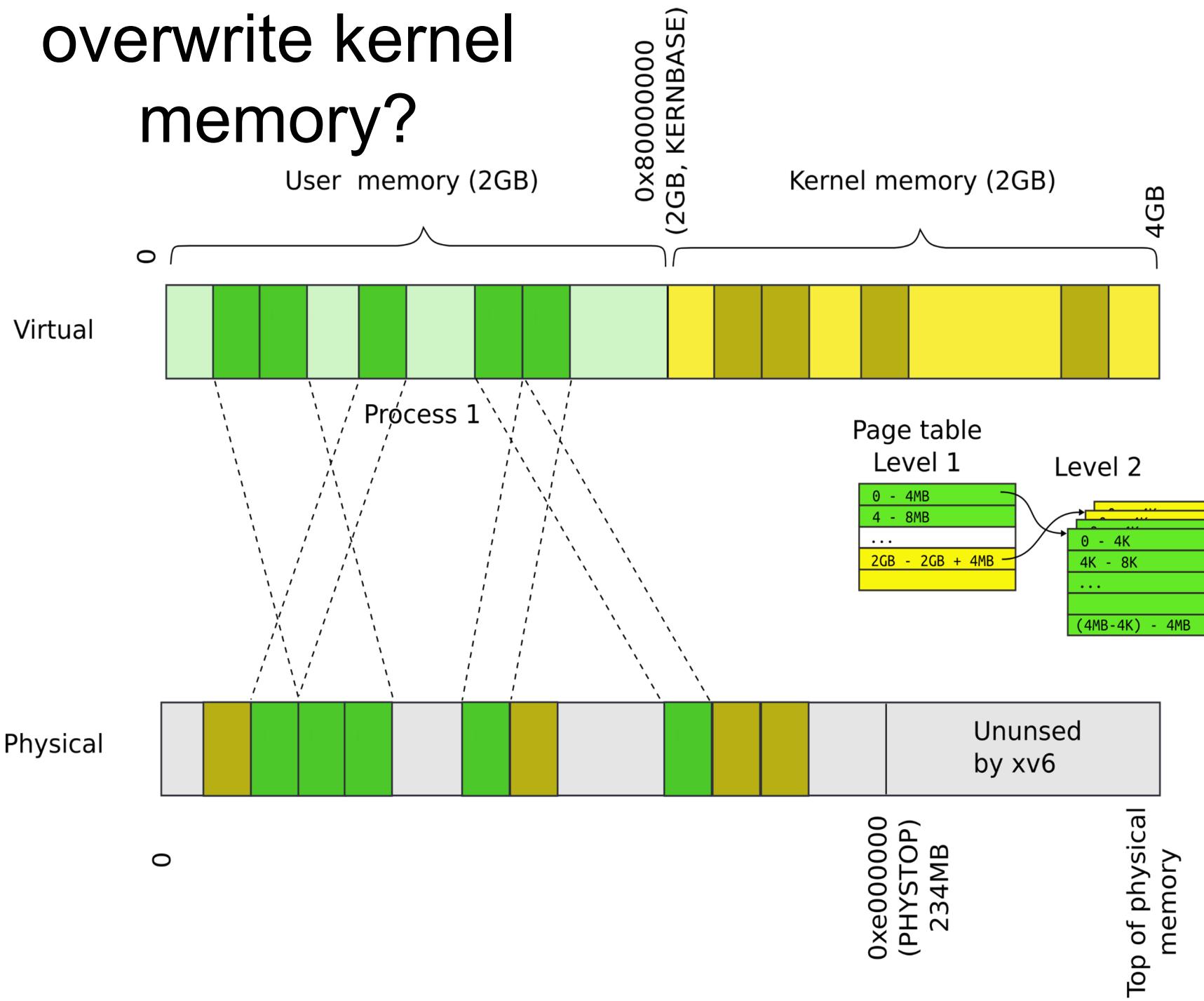
# Processing of interrupt (across PL)

- Need to change privilege level...

Detour:

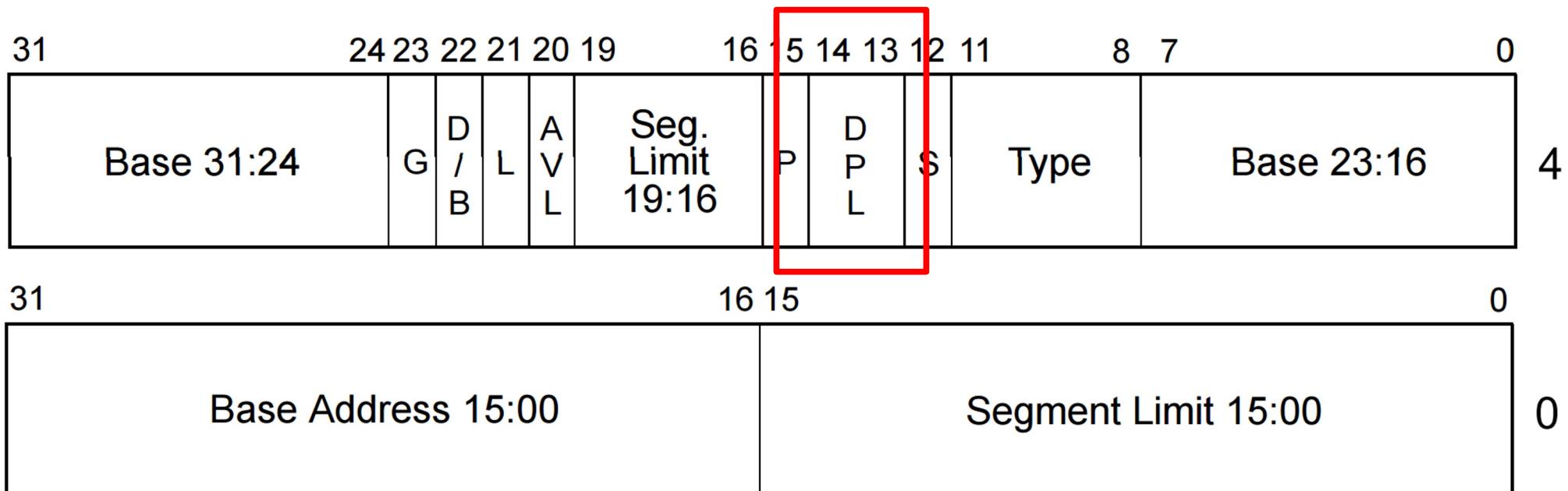
What are those privilege levels?

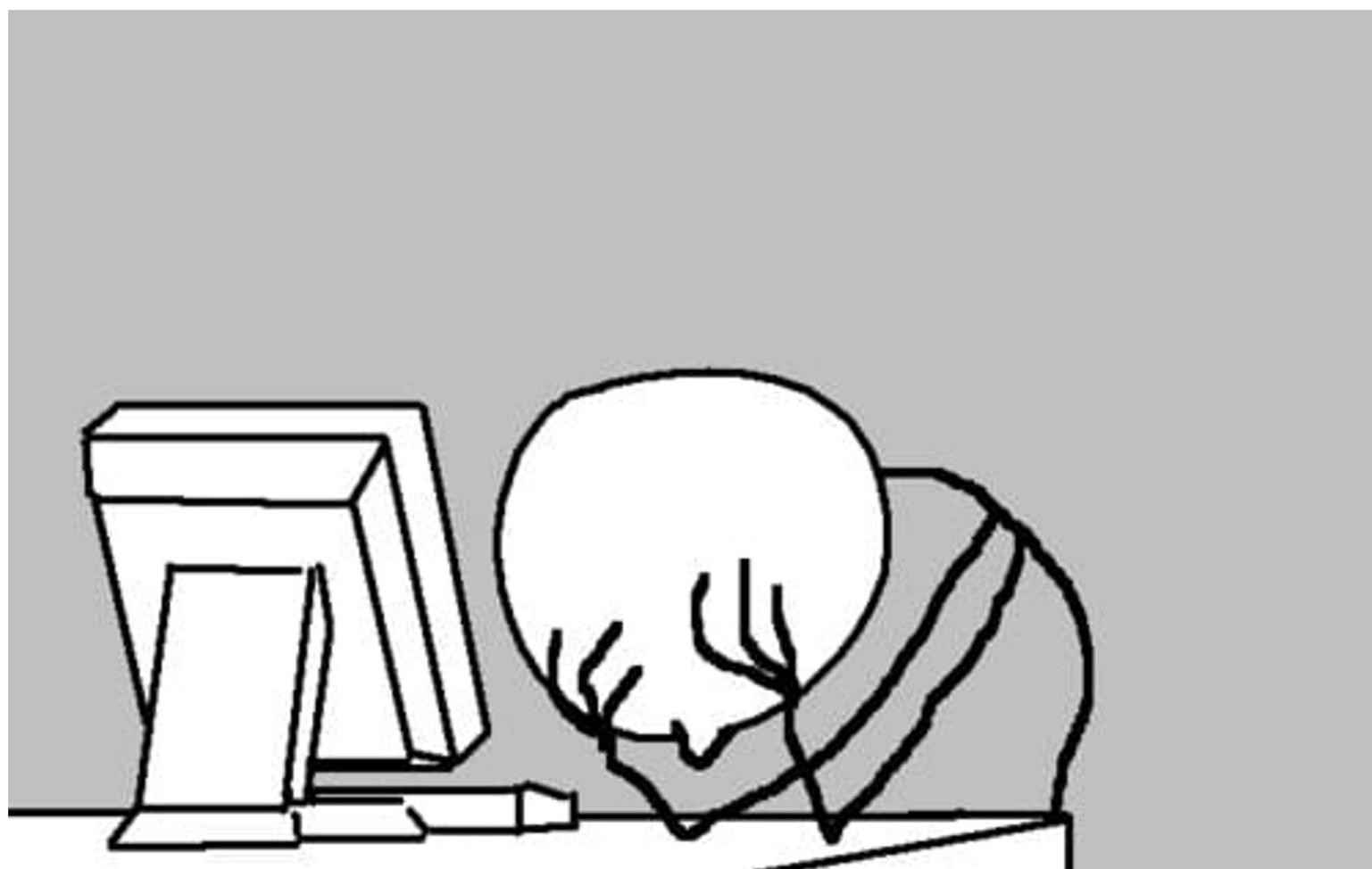
# Recap: Can a process overwrite kernel memory?

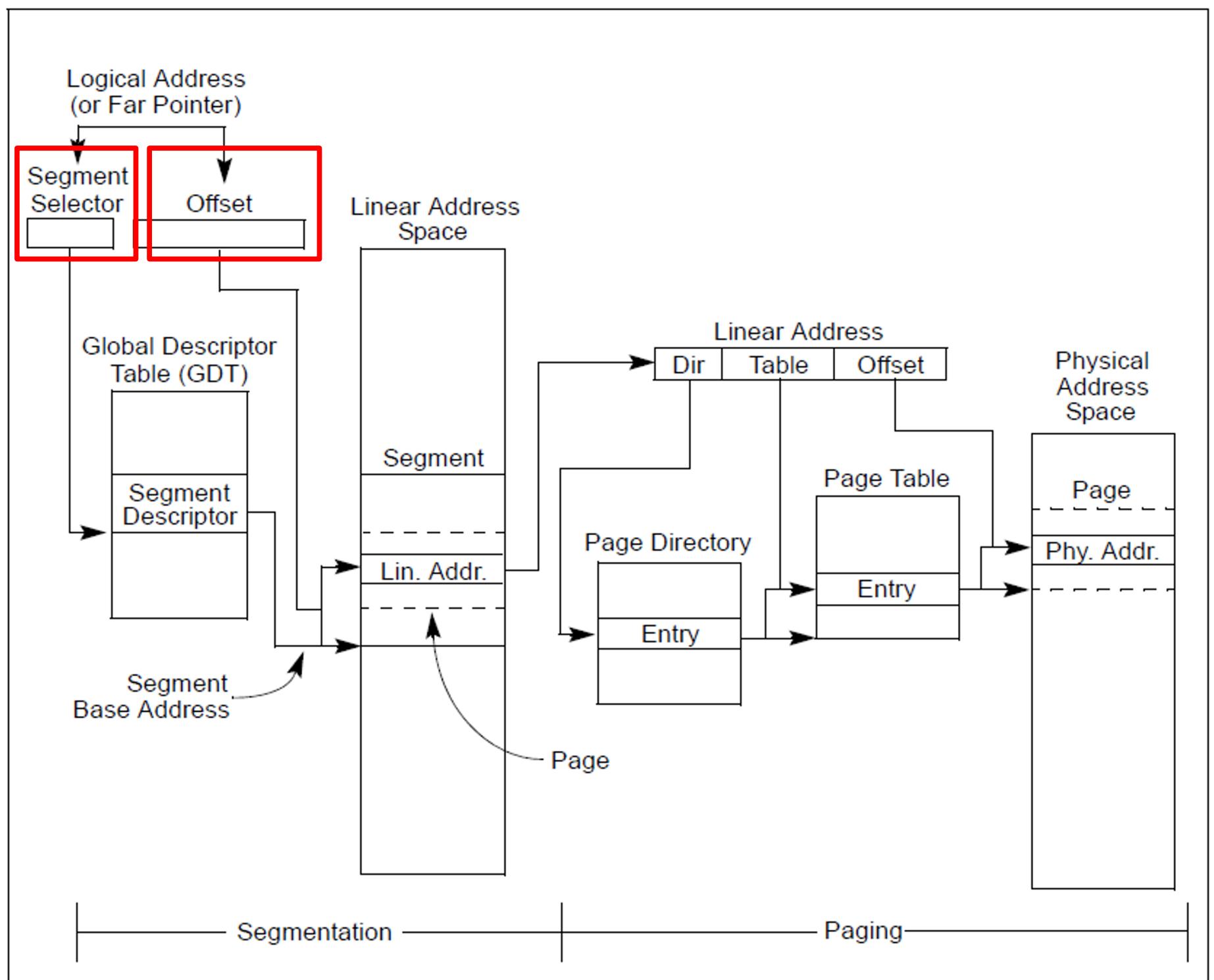


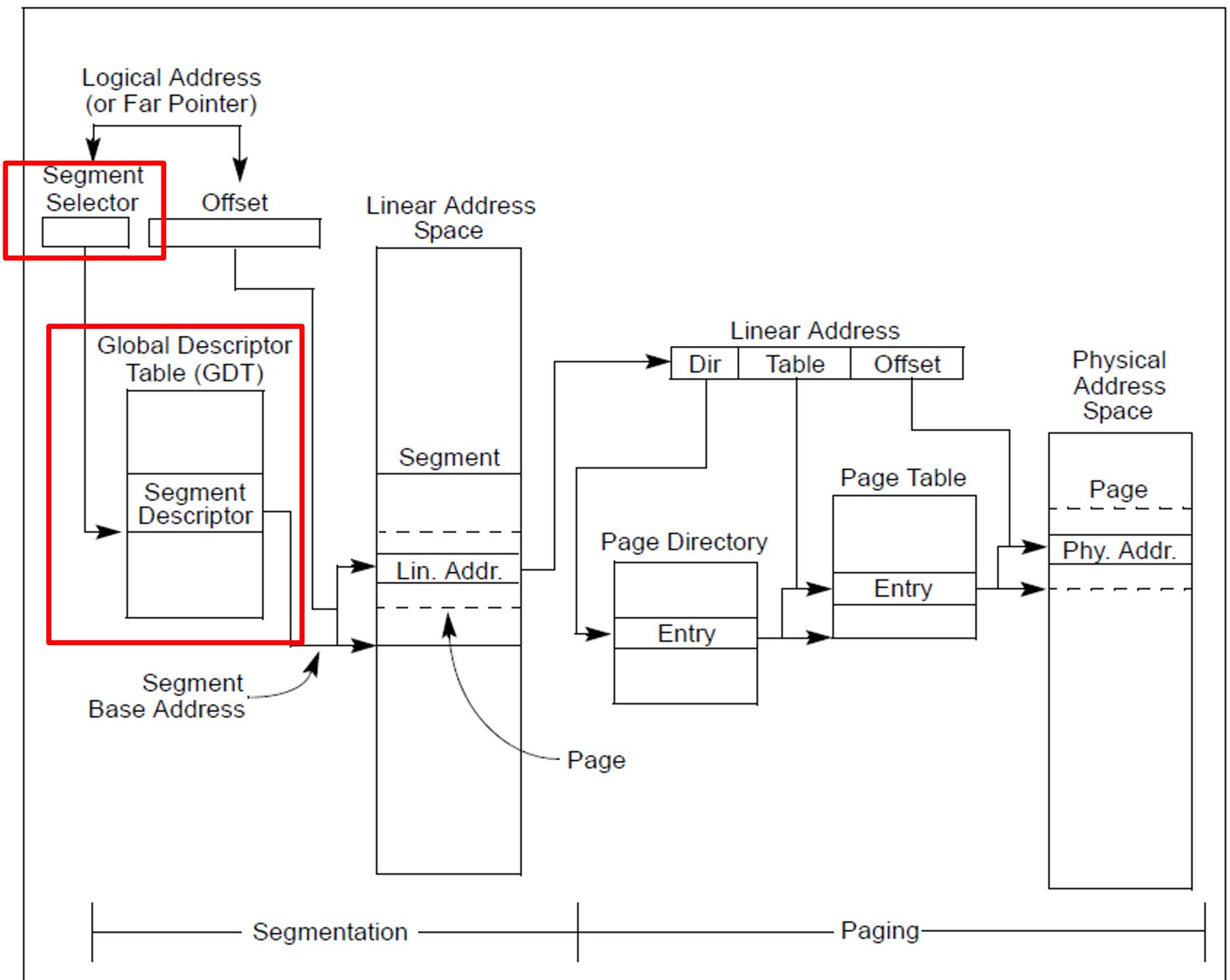
# Privilege levels

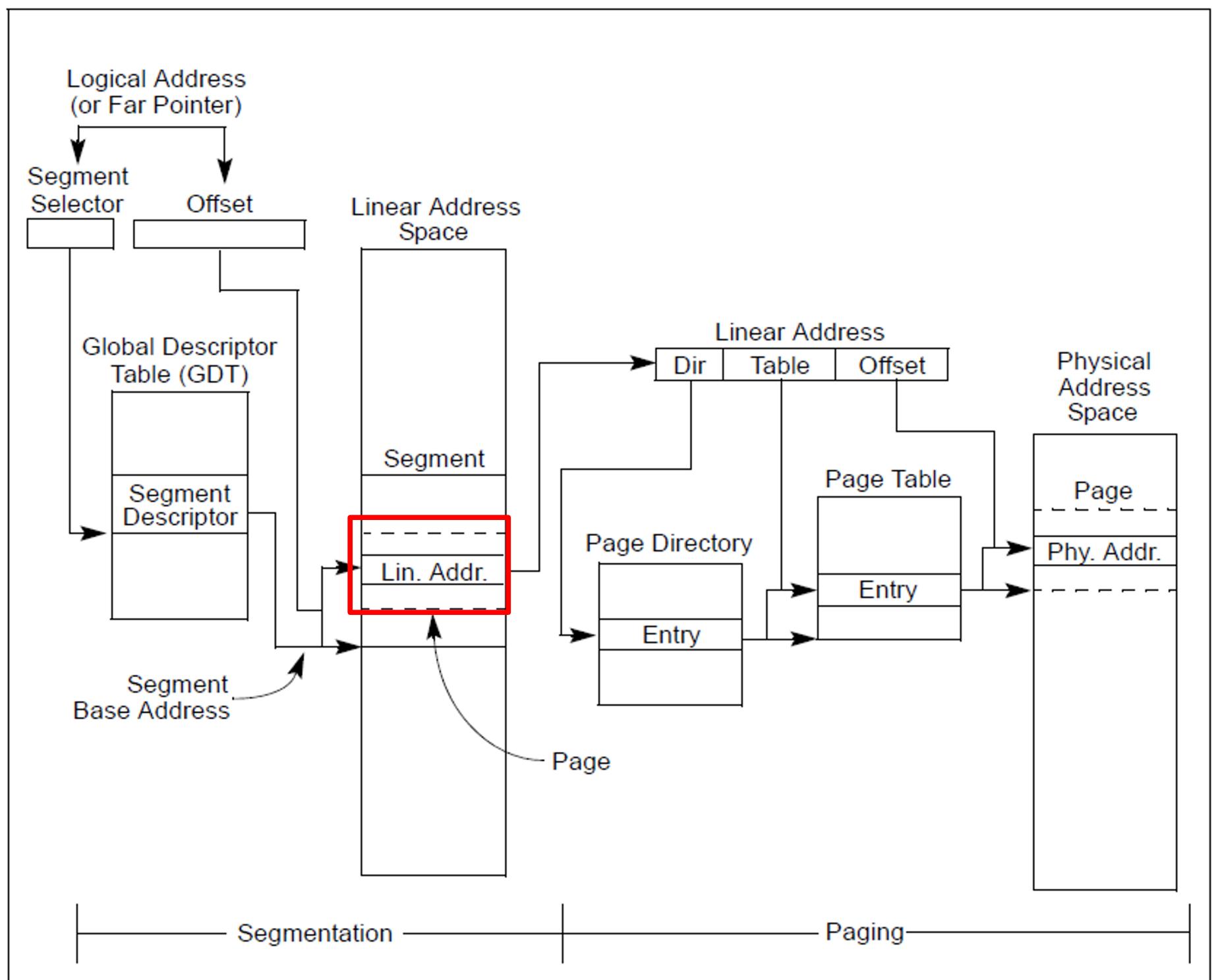
- Each segment has a **privilege level**
- **DPL** (descriptor privilege level)
- **4 privilege levels ranging 0-3**





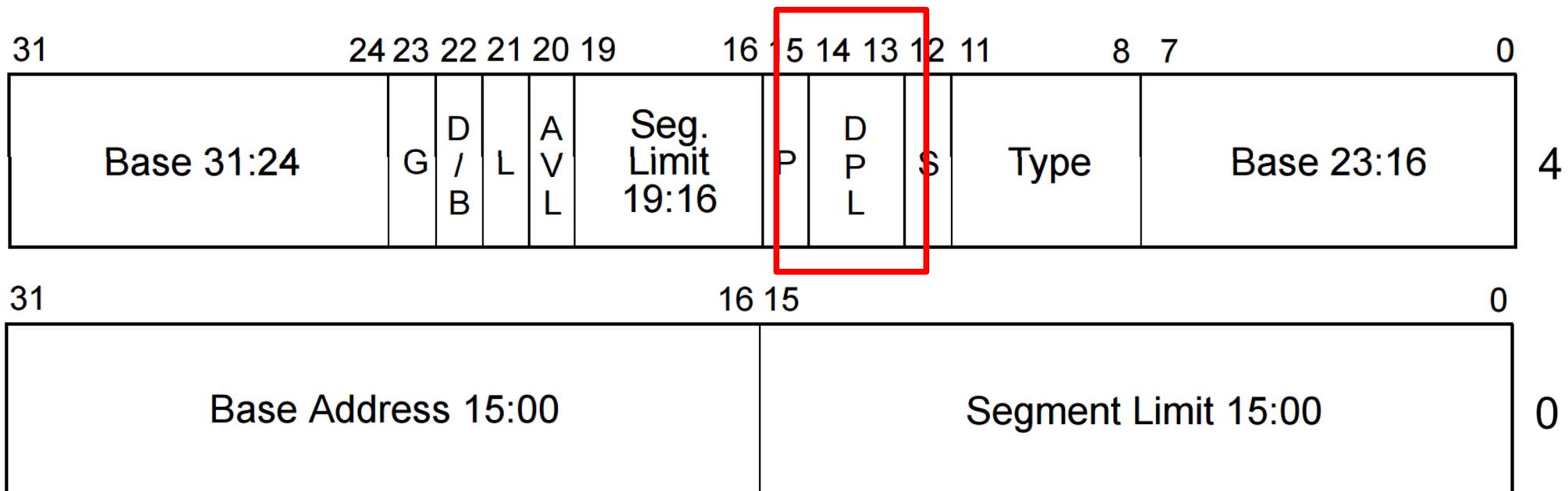






# Privilege levels

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



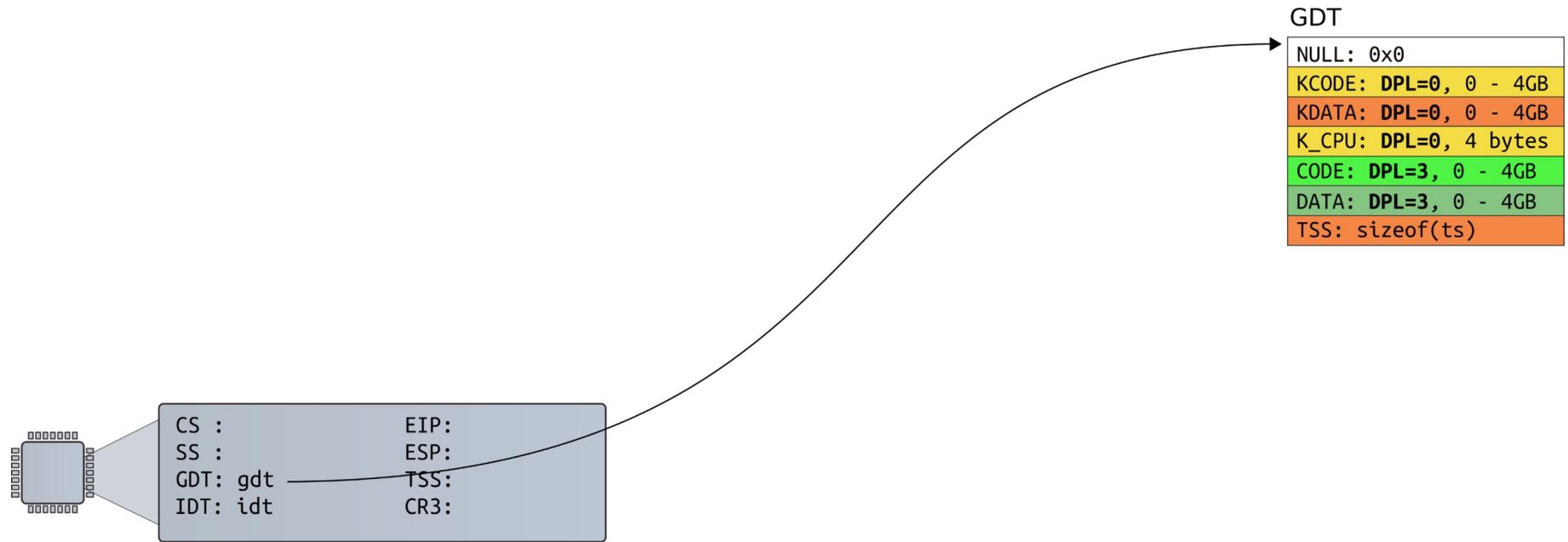
# Privilege levels

- 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

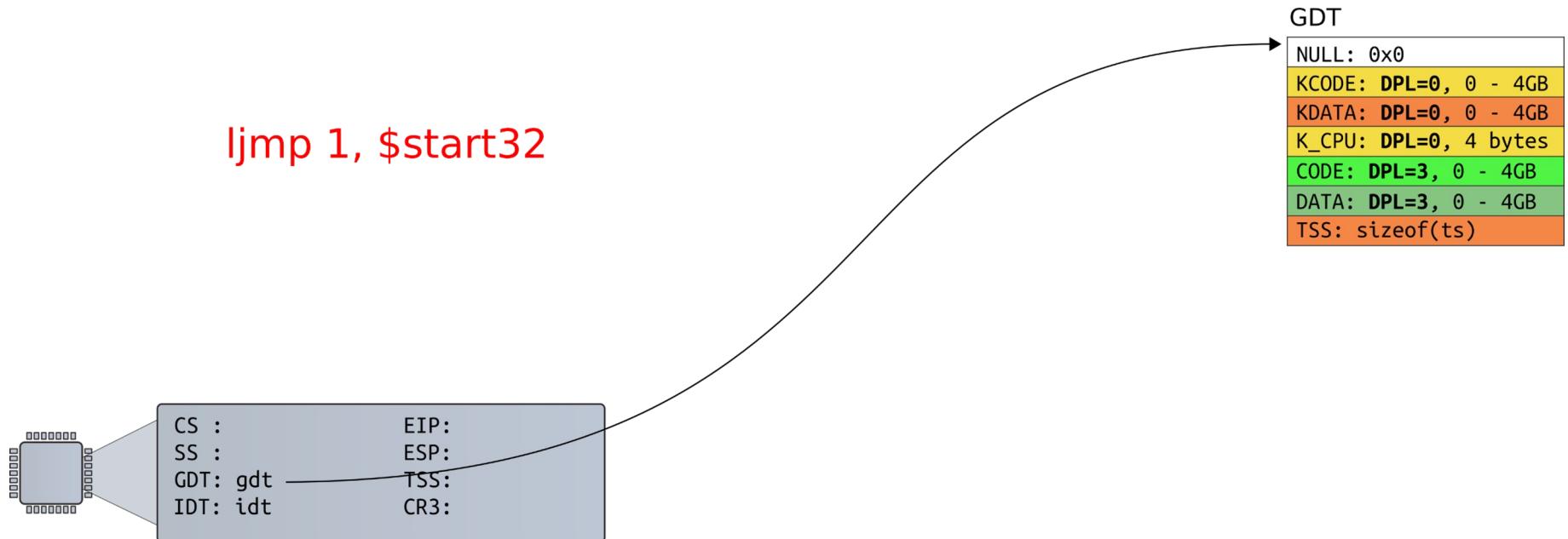
# Privilege level transitions

- 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)

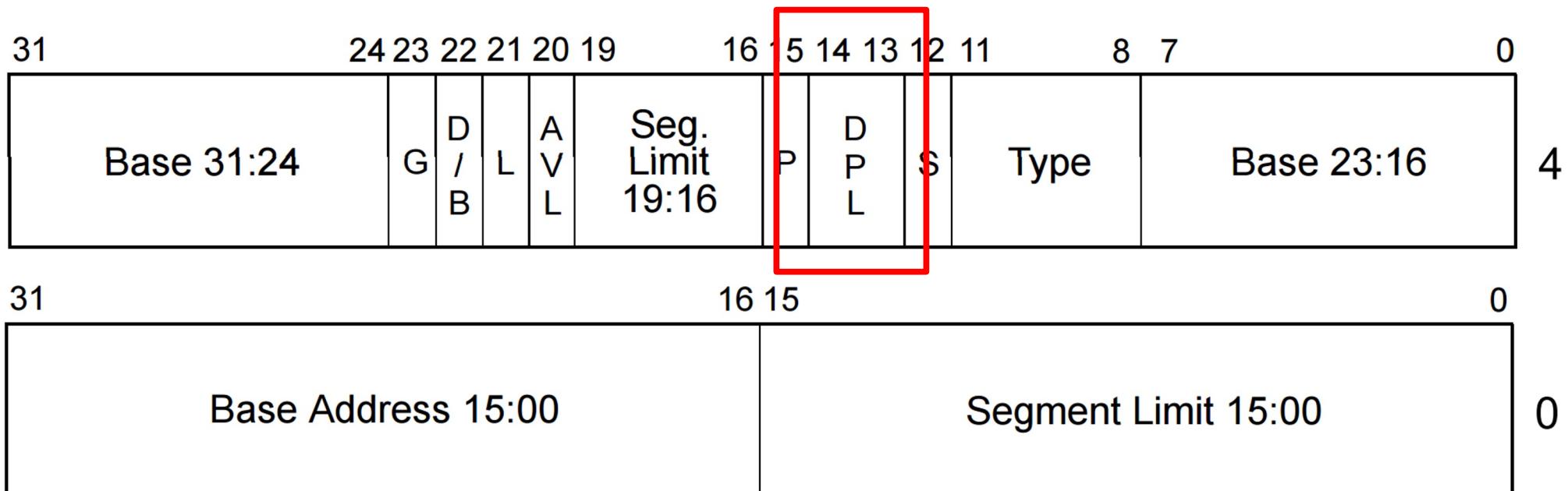


# Xv6 example: prepare to load GDT entry #1



# Privilege levels

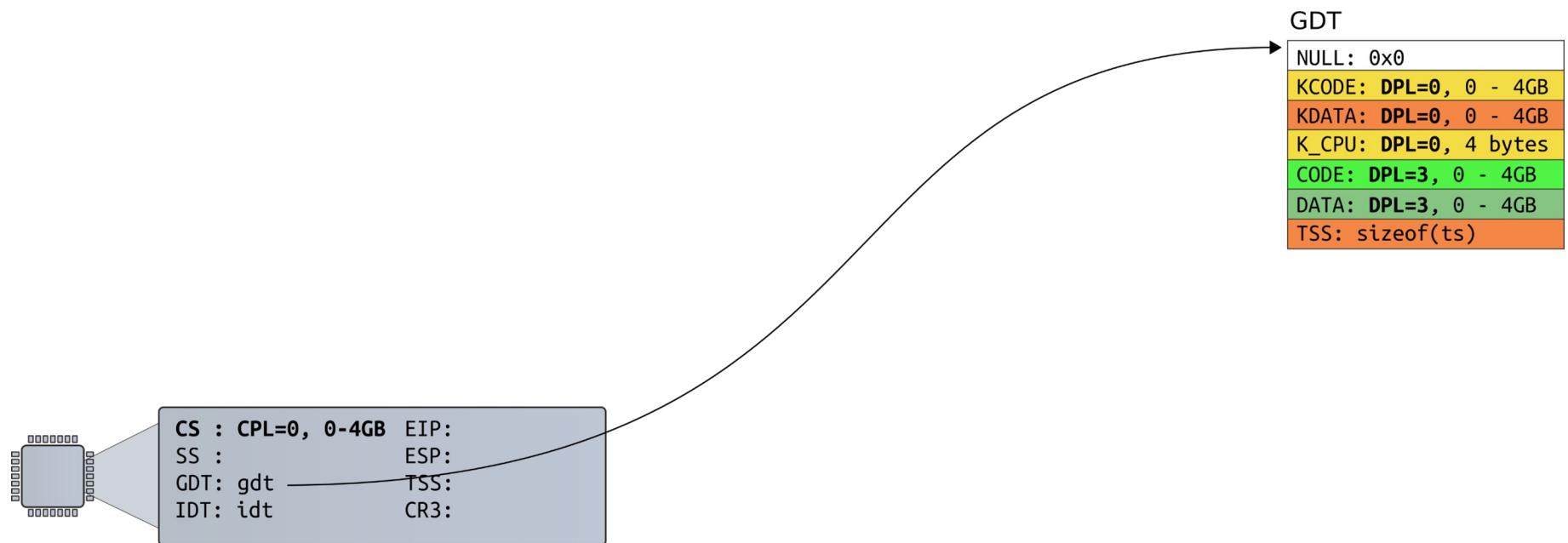
- Each segment has a **privilege level**
- **DPL** (descriptor privilege level)
- **4 privilege levels ranging 0-3**



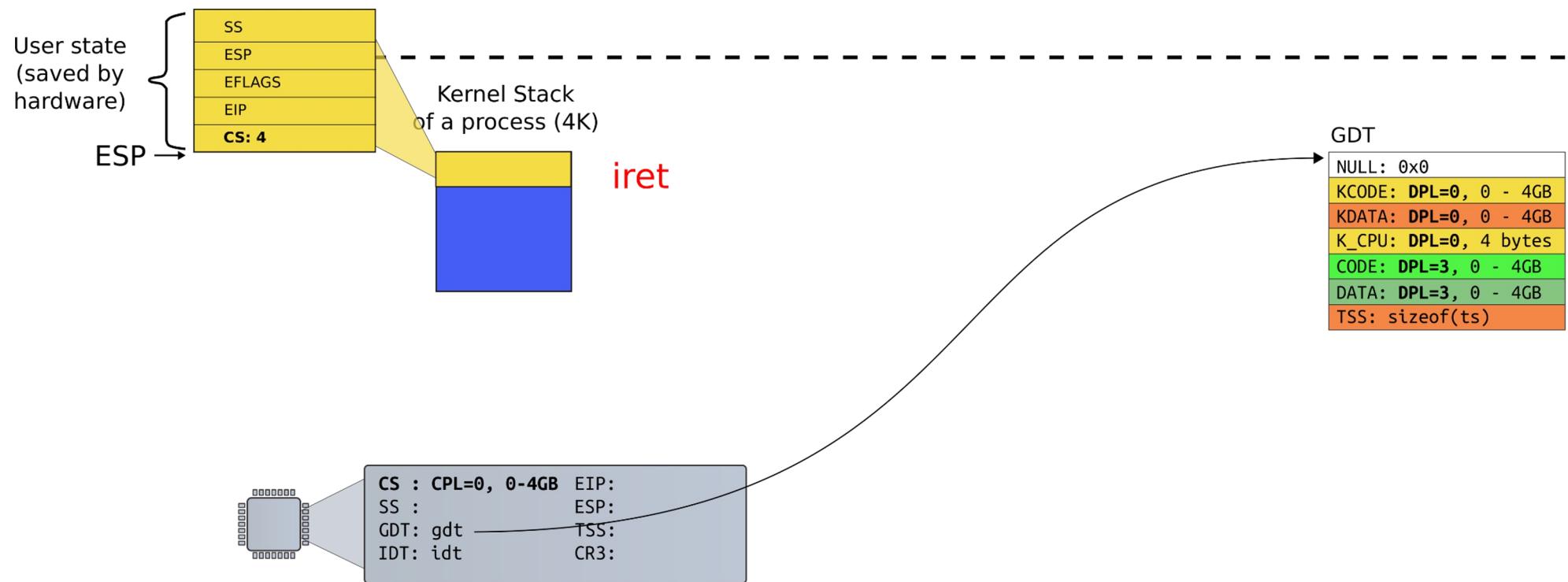
# How GDT is defined

```
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
```

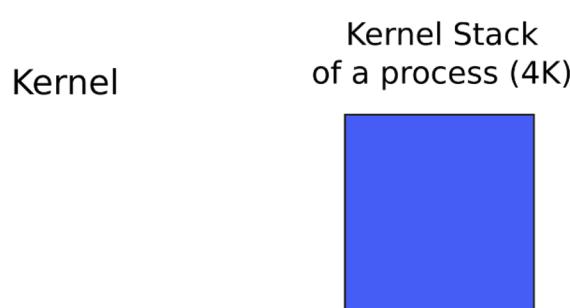
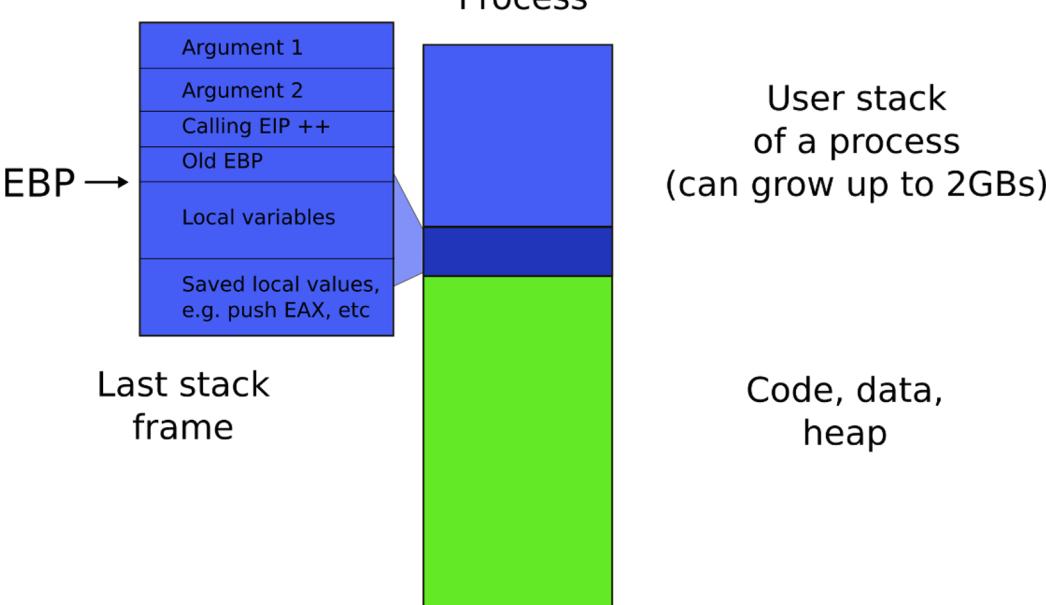
# Now CPL=0. We run in the kernel



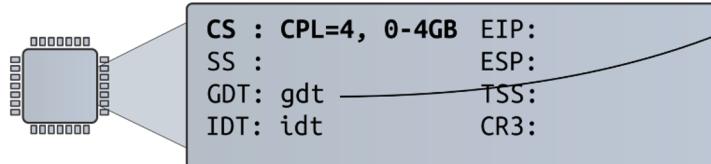
# iret: return to user, load GDT #4



# Run in user, CPL=3



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)



# 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

Poll: [PollEv.com/aburtsev](https://PollEv.com/aburtsev)

- How many privilege levels are there on x86?

# Poll: [PollEv.com/aburtsev](https://PollEv.com/aburtsev)

- Which privilege level is most privileged?

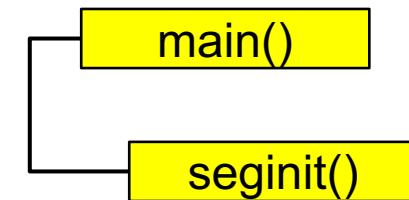
How GDT is initialized in xv6?

```
1317 main(void)
1318 {
1319     kinit1(end, P2V(4*1024*1024)); // phys page allocator
1320     kvmalloc(); // kernel page table
1321     mpinit(); // detect other processors
1322     lapicinit(); // interrupt controller
1323     seginit(); // segment descriptors
1324     cprintf("\ncpu%d: starting xv6\n\n", cpunum());
1325     picinit(); // another interrupt controller
1326     ioapicinit(); // another interrupt controller
1327     consoleinit(); // console hardware
1328     uartinit(); // serial port
1329     pinit(); // process table
1330     tvinit(); // trap vectors
...
}
```

main()

# Initialize GDT

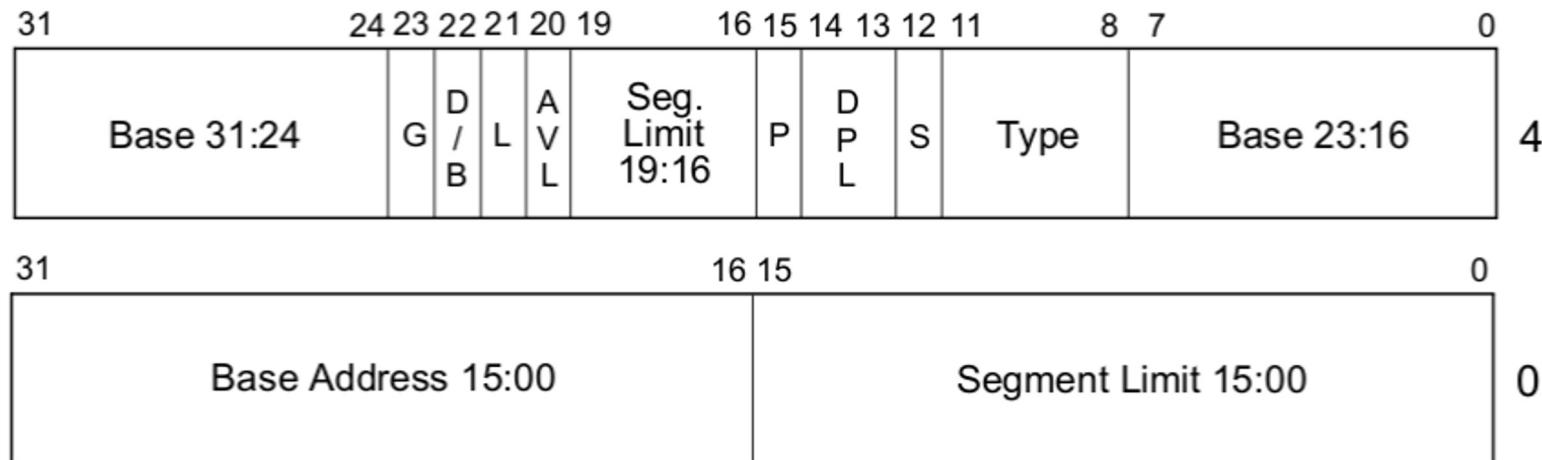
```
1712 // Set up CPU's kernel segment descriptors.  
1713 // Run once on entry on each CPU.  
1714 void  
1715 seginit(void)  
1716 {  
1717     struct cpu *c;  
  
...  
1723     c = &cpus[cpuid()];  
1724     c->gdt[SEG_KCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, 0);  
1725     c->gdt[SEG_KDATA] = SEG(STA_W, 0, 0xffffffff, 0);  
1726     c->gdt[SEG_UCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, DPL_USER);  
1727     c->gdt[SEG_UDATA] = SEG(STA_W, 0, 0xffffffff, DPL_USER);  
1728     lgdt(c->gdt, sizeof(c->gdt));  
1729 }
```



# Struct CPU

```
2300 // Per-CPU state
2301 struct cpu {
2302     uchar apicid;                      // Local APIC ID
2303     struct context *scheduler;          // swtch() here to enter scheduler
2304     struct taskstate ts;               // Used by x86 to find stack for interrupt
2305     struct segdesc gdt[NSEGS];         // x86 global descriptor table
2306     volatile uint started;             // Has the CPU started?
2307     int ncli;                         // Depth of pushcli nesting.
2308     int intena;                       // Were interrupts enabled before pushcli?
2309     struct proc *proc;                // The process running on this cpu or null
2310 };
2311
2312 extern struct cpu cpus[NCPU];
```

# Segment descriptor (entry in GDT)



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

# Segment Descriptor

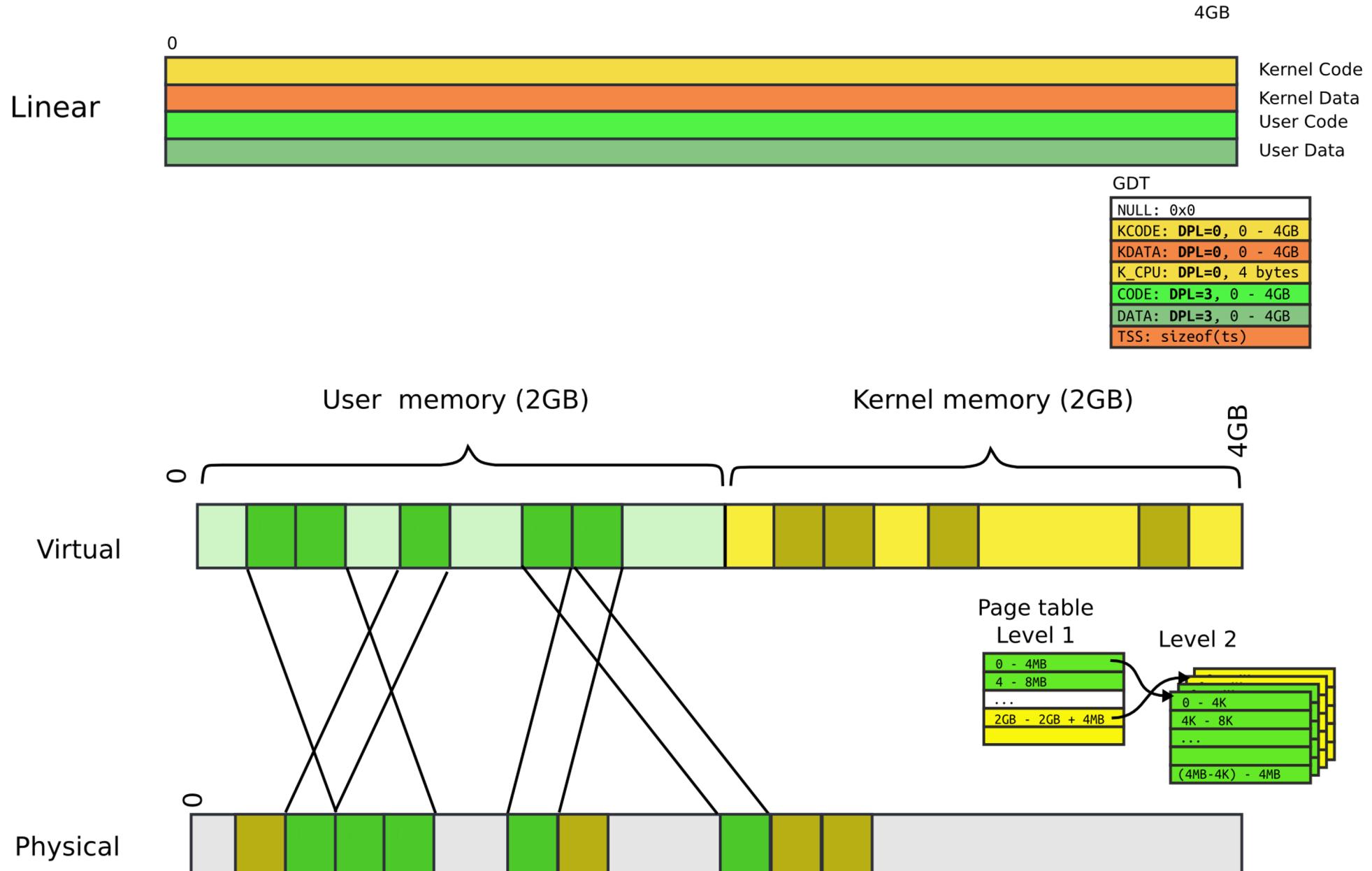
```
0724 // Segment Descriptor
0725 struct segdesc {
0726     uint lim_15_0 : 16;    // Low bits of segment limit
0727     uint base_15_0 : 16;  // Low bits of segment base address
0728     uint base_23_16 : 8; // Middle bits of segment base address
0729     uint type : 4;      // Segment type (see STS_ constants)
0730     uint s : 1;          // 0 = system, 1 = application
0731     uint dpl : 2;        // Descriptor Privilege Level
0732     uint p : 1;          // Present
0733     uint lim_19_16 : 4;  // High bits of segment limit
0734     uint avl : 1;        // Unused (available for software use)
0735     uint rsv1 : 1;       // Reserved
0736     uint db : 1;         // 0 = 16-bit segment, 1 = 32-bit segment
0737     uint g : 1;          // Granularity: limit scaled by 4K when set
0738     uint base_31_24 : 8; // High bits of segment base address
0739 };
```

# 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 do we protect the kernel?**

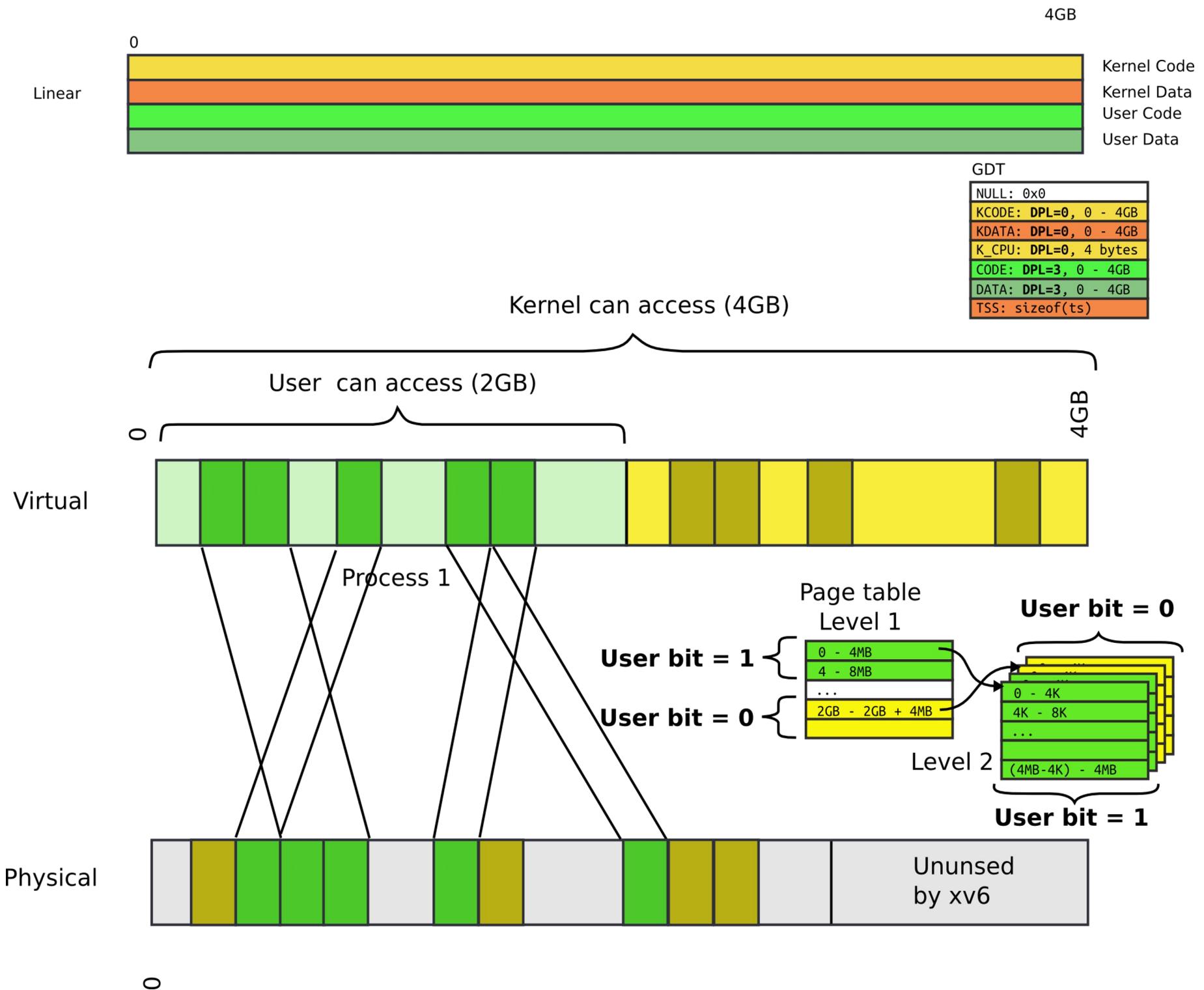
# Real world

- 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 do we protect the kernel?
- Page tables



# 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



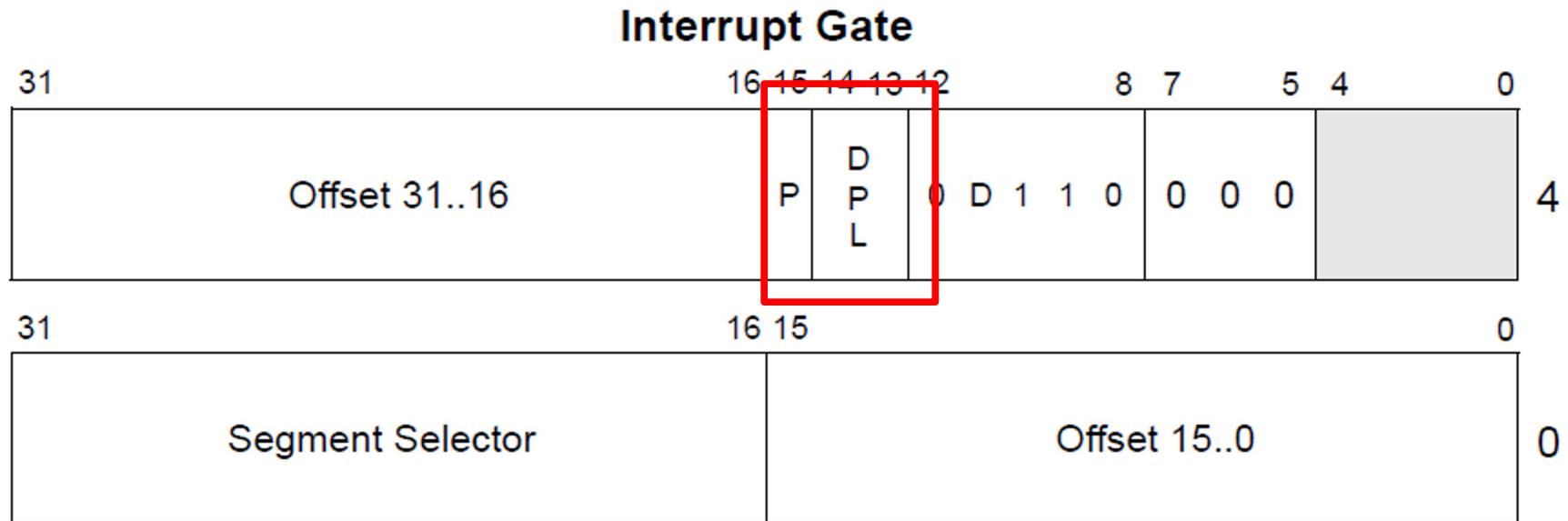
End of detour:  
Back to handling interrupts

Processing of an interrupt when change of a  
privilege level is required

# Processing of interrupt (across PL)

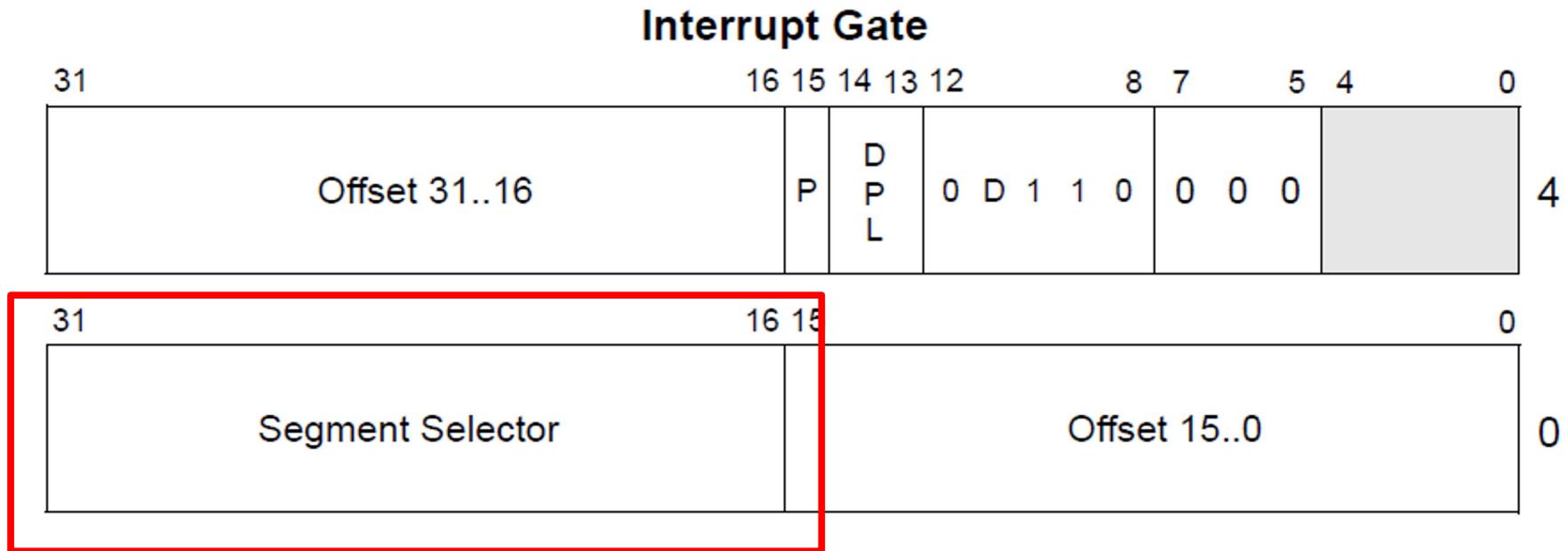
- Assume we're at CPL =3 (user)

# Interrupt descriptor (an entry in the IDT)



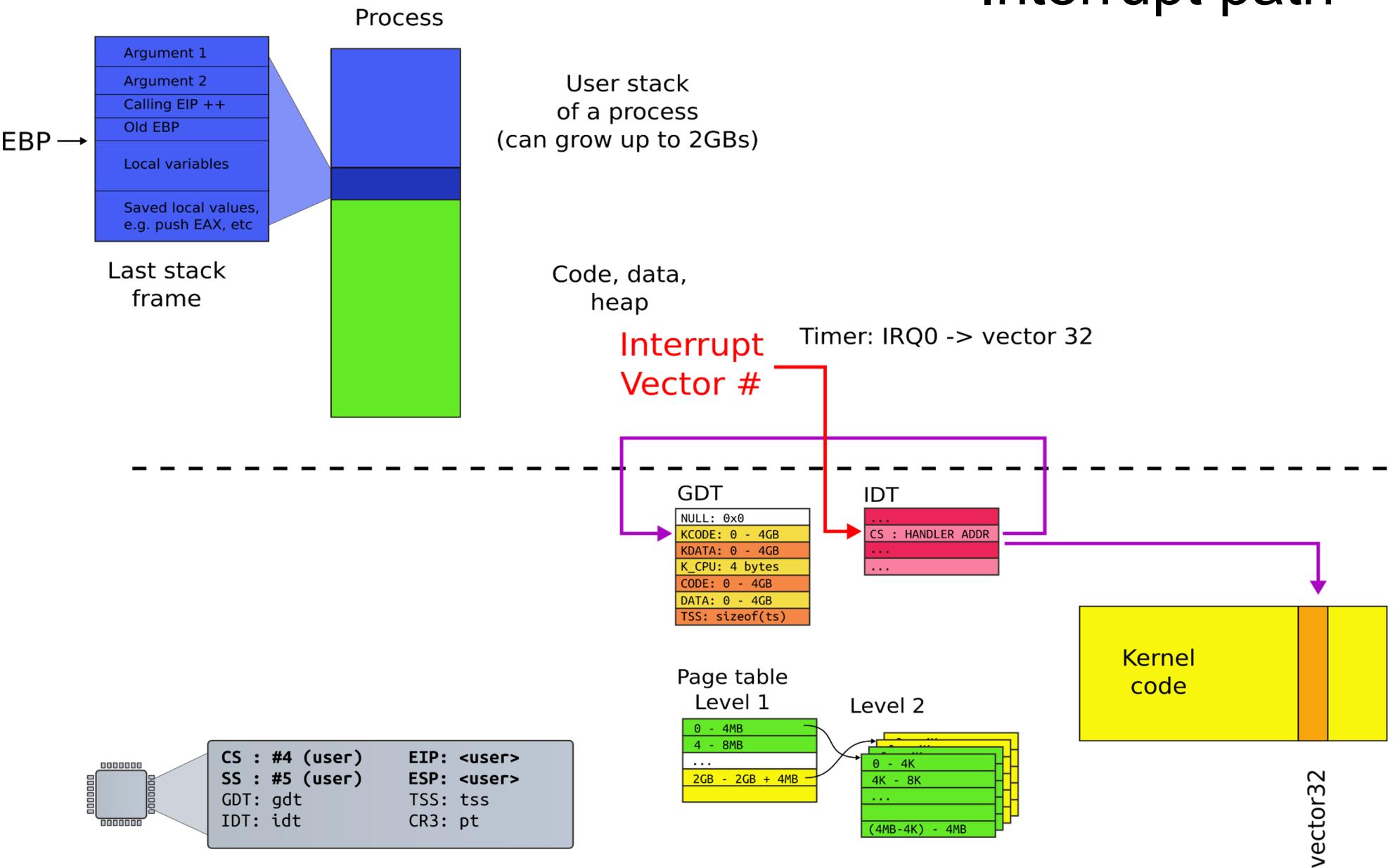
- Interrupt is allowed if...
  - current privilege level (CPL) is less or equal to descriptor privilege level (DPL)
- User cannot invoke `int 0x32`

# Interrupt descriptor (an entry in the IDT)



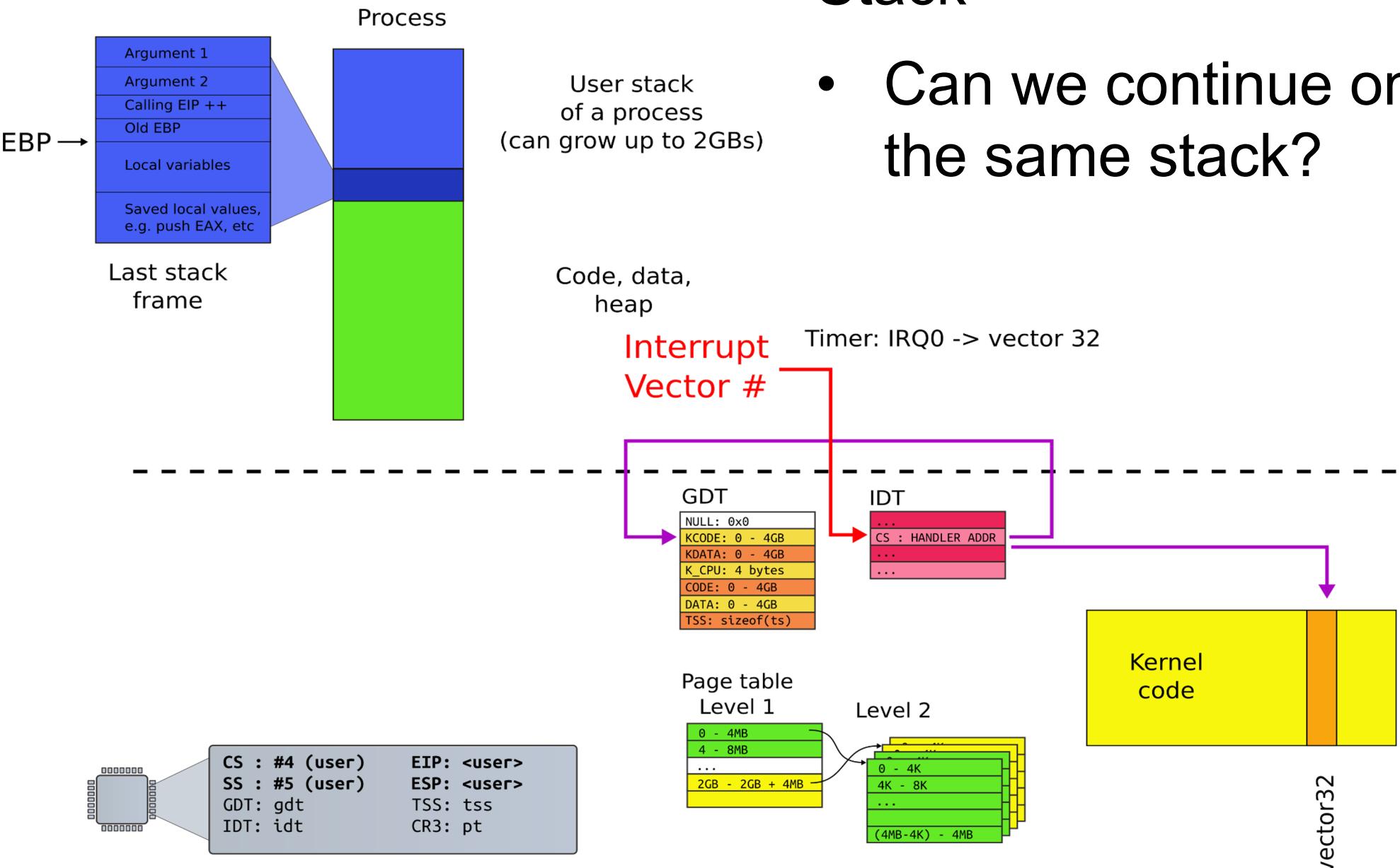
- This new segment can be more privileged
  - E.g.,  $CPL = 3$ ,  $DPL = 3$ , new segment can be  $PL = 0$
- This is how user-code ( $PL=3$ ) transitions into kernel ( $PL=0$ )

# Interrupt path



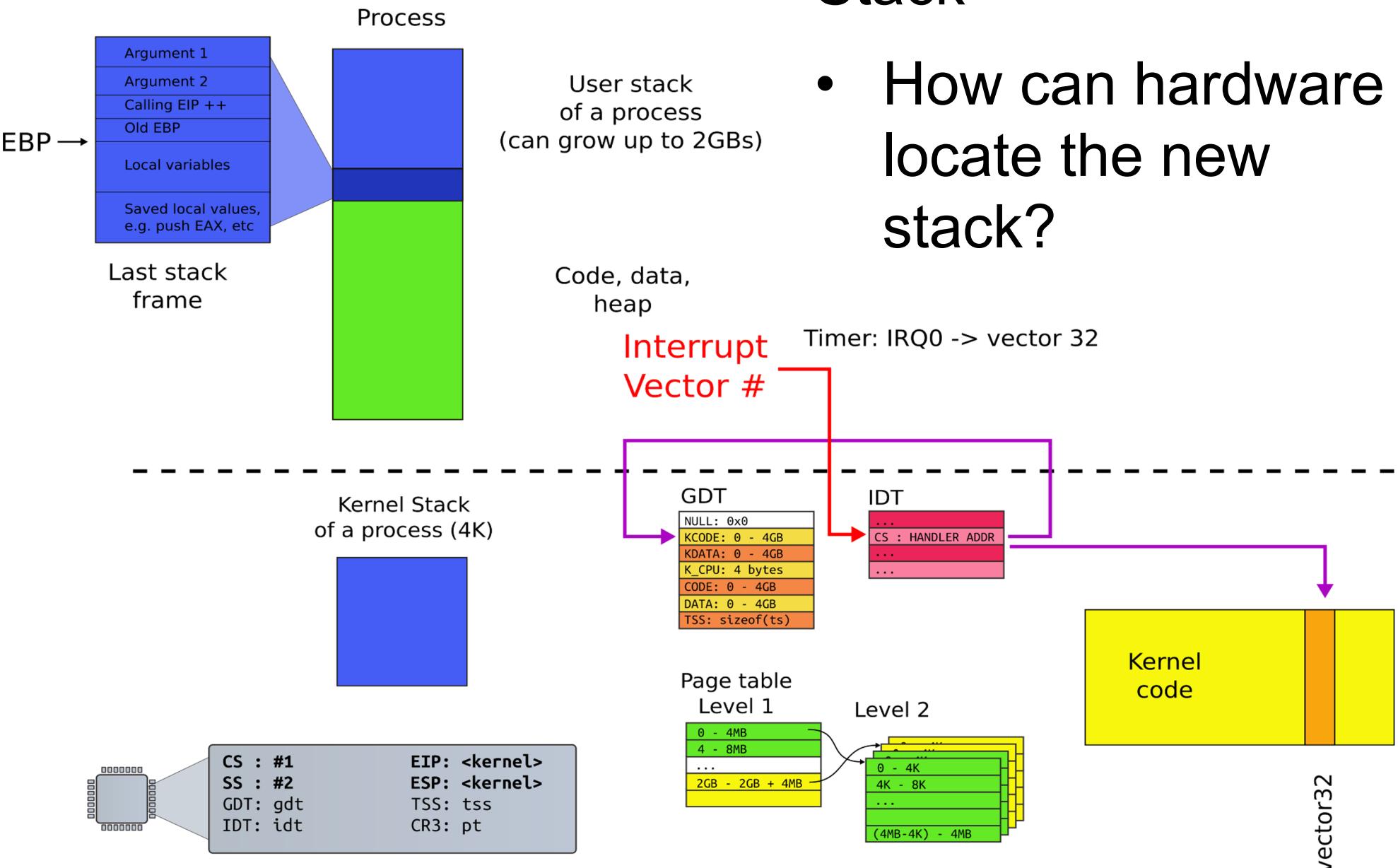
# Stack

- Can we continue on the same stack?

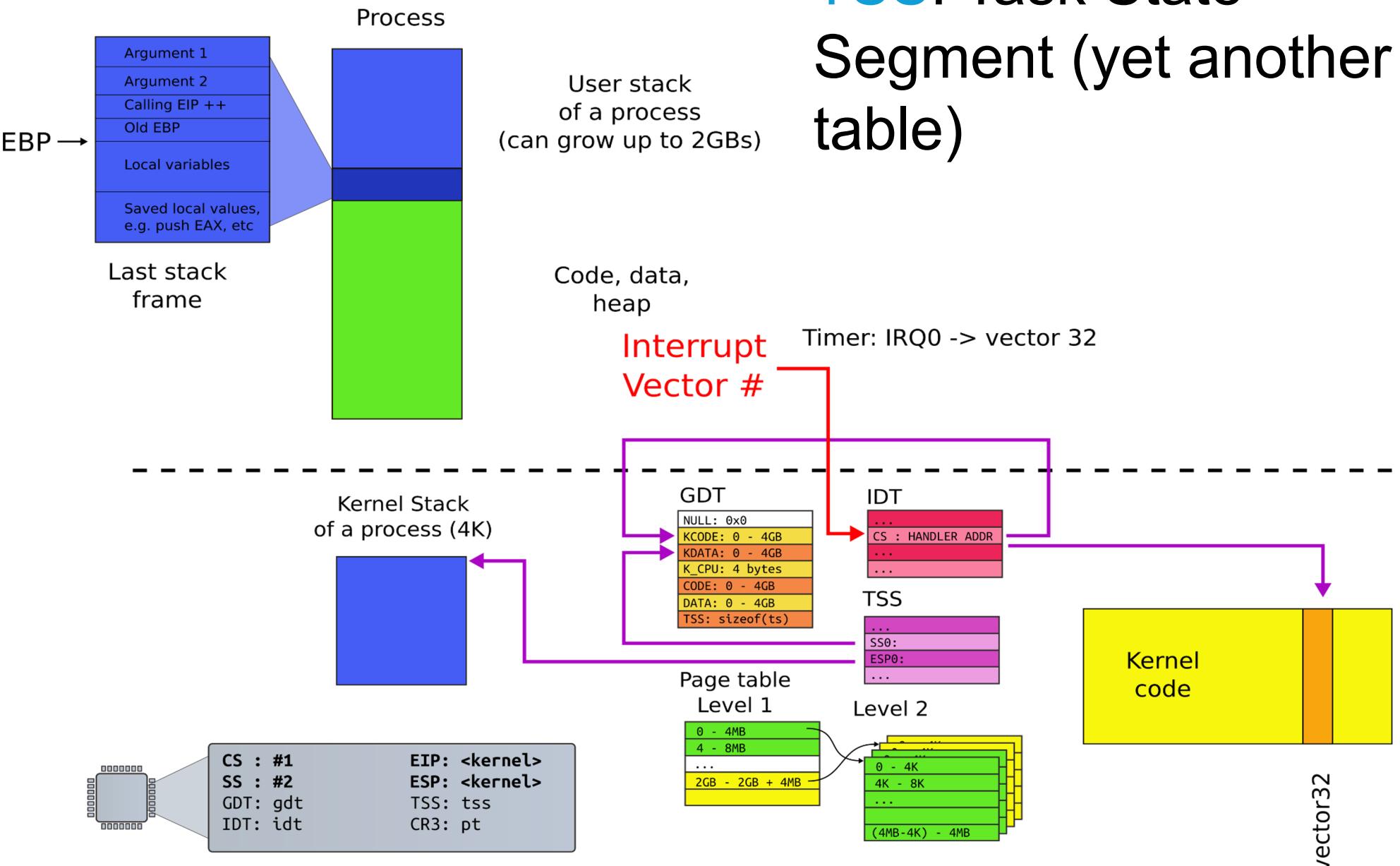


# Stack

- How can hardware locate the new stack?



# 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

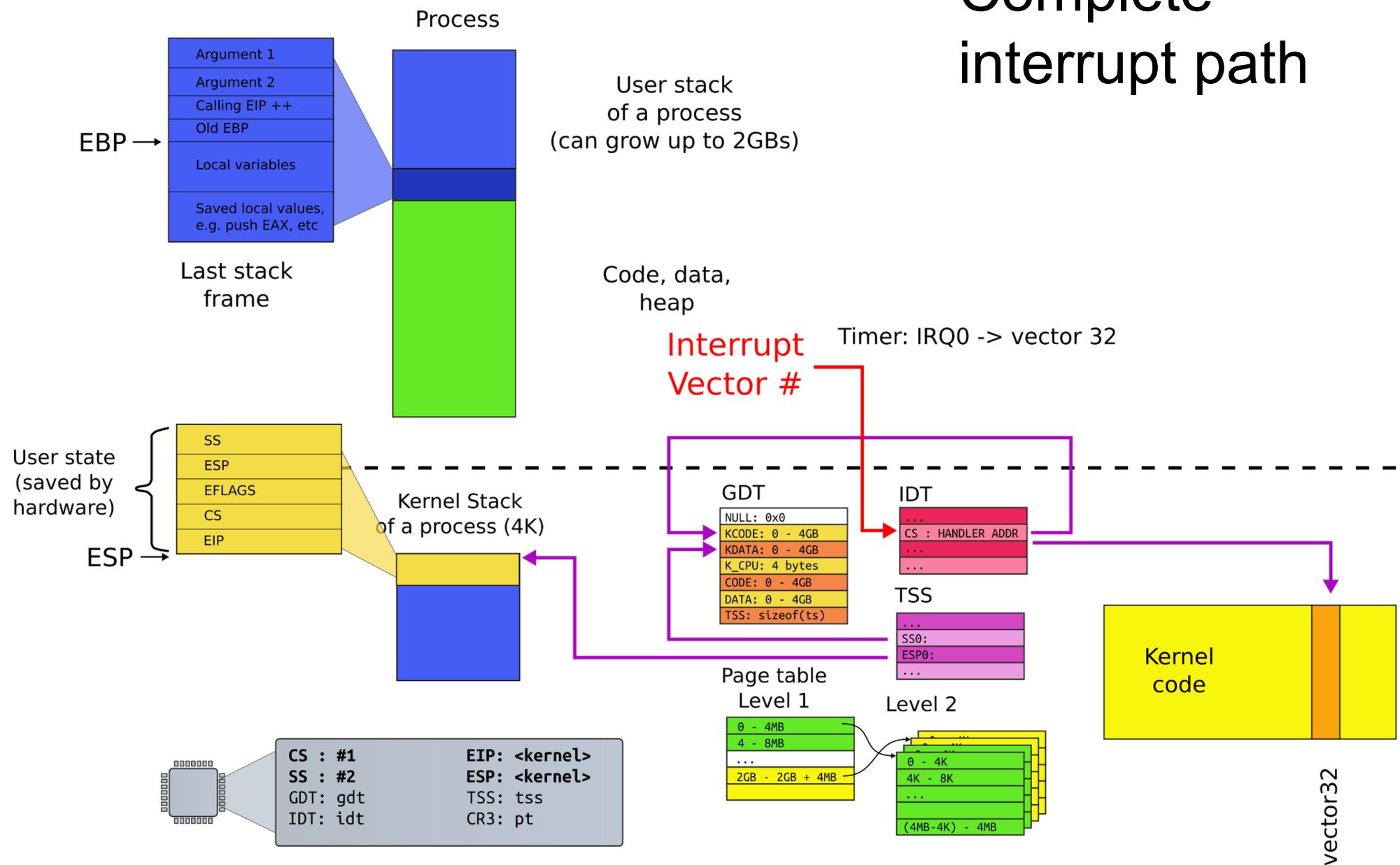
# Processing of interrupt (across PL)

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 interrupts enabled EFLAGS bit
6. Begin execution of a handler

# Poll: [PollEv.com/aburtsev](https://PollEv.com/aburtsev)

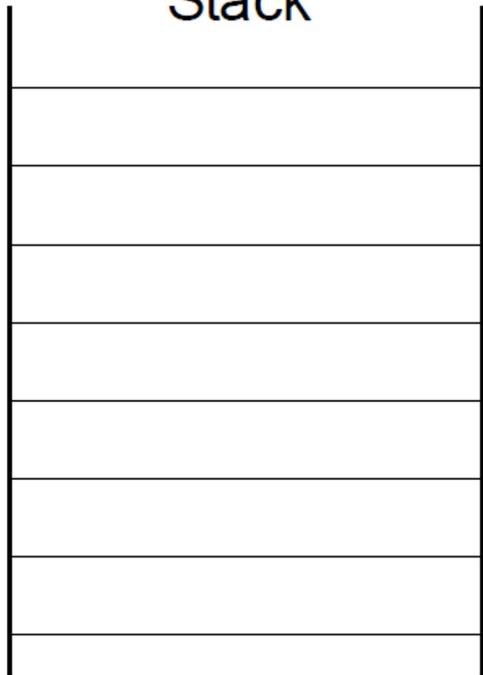
- Which registers are saved on cross-PL interrupt transition?

# Complete interrupt path



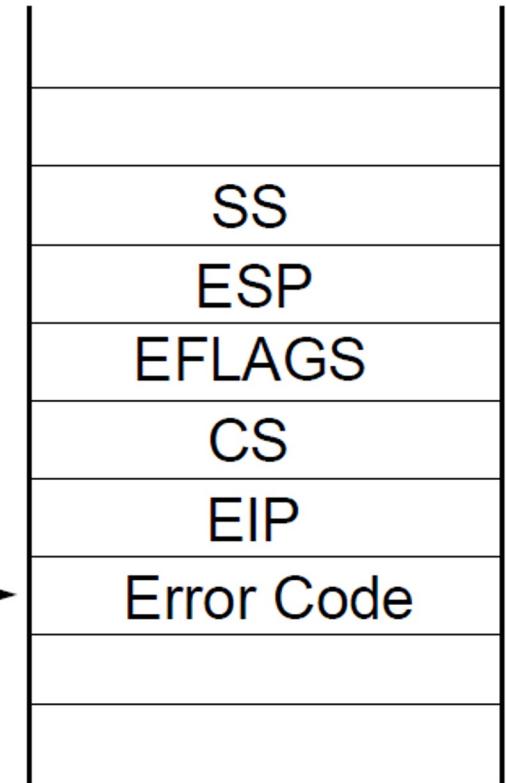
## Stack Usage with Privilege-Level Change

Interrupted Procedure's  
Stack



ESP Before  
Transfer to Handler

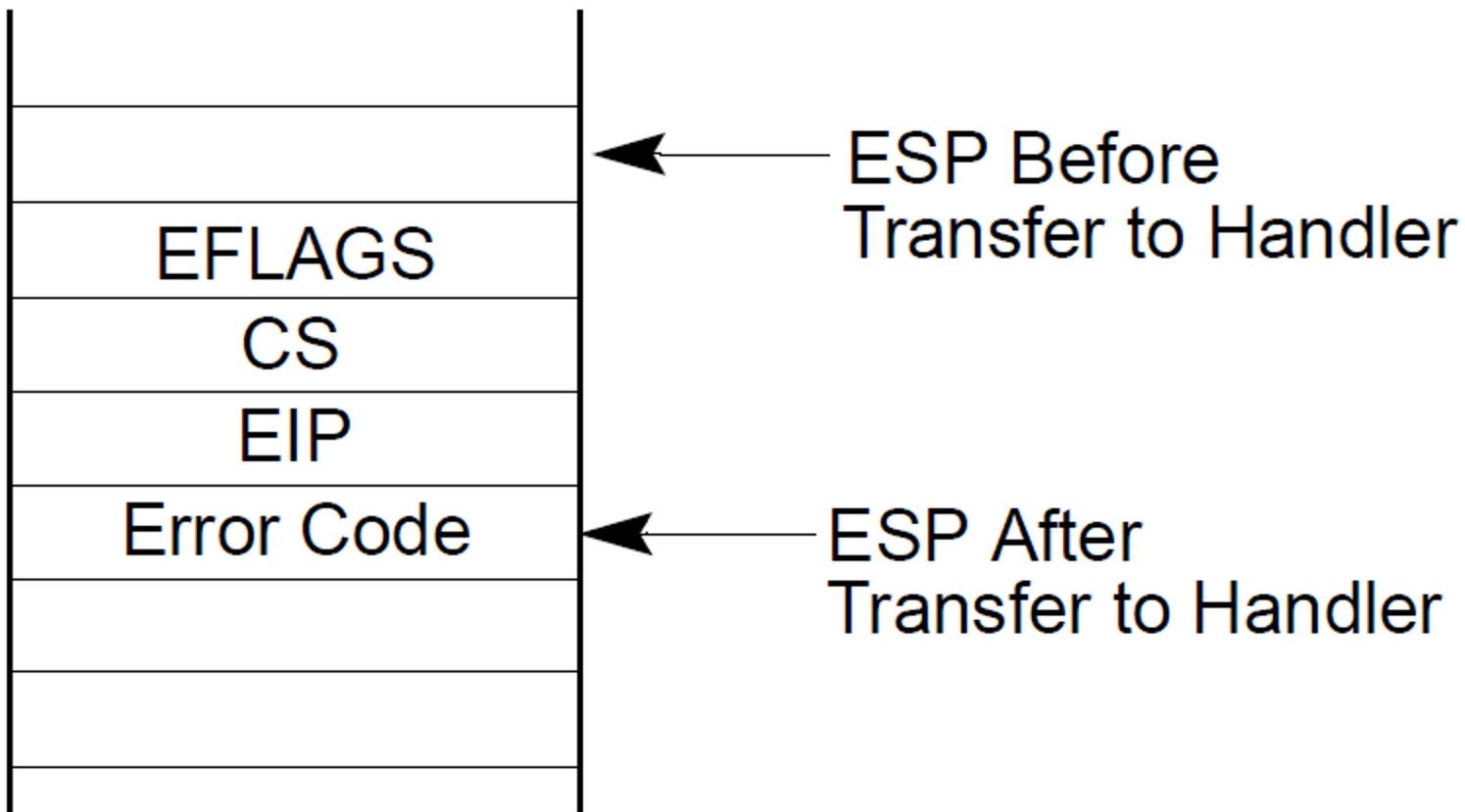
Handler's Stack



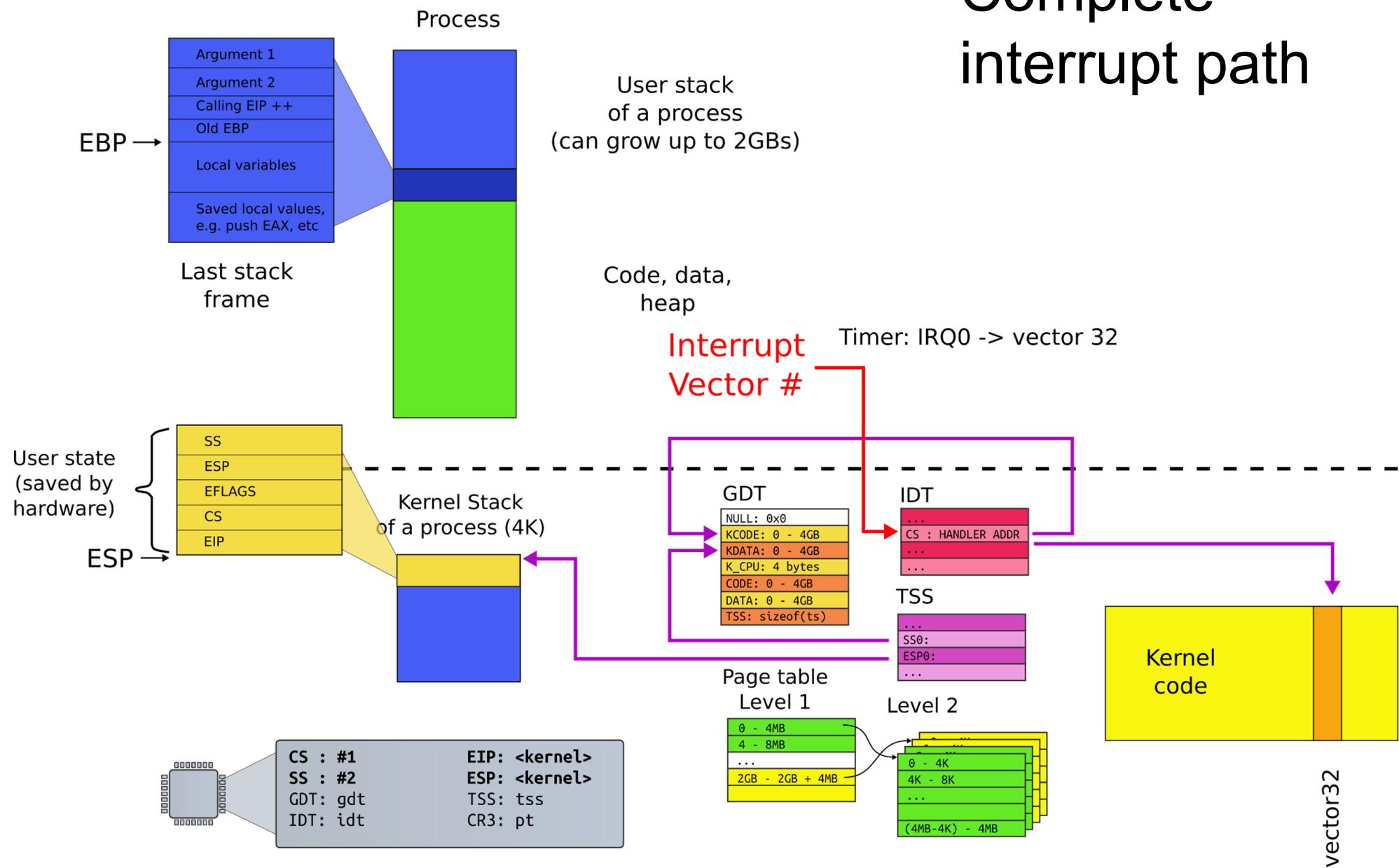
ESP After  
Transfer to Handler

## Stack Usage with No Privilege-Level Change

### Interrupted Procedure's and Handler's Stack



# 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. <sup>1</sup>
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. <sup>2</sup>
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. <sup>3</sup>
18	#MC	Machine Check	Error codes (if any) and source are model dependent. <sup>4</sup>
19	#XM	SIMD Floating-Point Exception	SIMD Floating-Point Instruction <sup>5</sup>
20-31		Reserved	
32-255		Maskable Interrupts	External interrupt from INTR pin or INT <i>n</i> 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. <sup>1</sup>
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. <sup>2</sup>
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. <sup>3</sup>
18	#MC	Machine Check	Error codes (if any) and source are model dependent. <sup>4</sup>
19	#XM	SIMD Floating-Point Exception	SIMD Floating-Point Instruction <sup>5</sup>
20-31		Reserved	
32-255		Maskable Interrupts	External interrupt from INTR pin or INT <i>n</i> 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	#DR	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. <sup>1</sup>
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. <sup>2</sup>
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. <sup>3</sup>
18	#MC	Machine Check	Error codes (if any) and source are model dependent. <sup>4</sup>
19	#XM	SIMD Floating-Point Exception	SIMD Floating-Point Instruction <sup>5</sup>
20-31		Reserved	
32-255		Maskable Interrupts	External interrupt from INTR pin or INT <i>n</i> 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

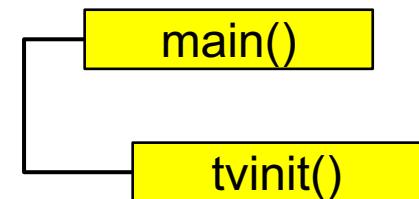
Xv6 source

```
1317 main(void)
1318 {
1319     kinit1(end, P2V(4*1024*1024)); // phys page allocator
1320     kvmalloc(); // kernel page table
1321     mpinit(); // detect other processors
1322     lapicinit(); // interrupt controller
1323     seginit(); // segment descriptors
1324     cprintf("\ncpu%d: starting xv6\n\n", cpunum());
1325     picinit(); // another interrupt controller
1326     ioapicinit(); // another interrupt controller
1327     consoleinit(); // console hardware
1328     uartinit(); // serial port
1329     pinit(); // process table
1330     tvinit(); // trap vectors
1331     binit(); // buffer cache
...
}
```

main()

```
3316 void
3317 tvinit(void)
3318 {
3319     int i;
3320
3321     for(i = 0; i < 256; i++)
3322         SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
3323     SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3,
3324                                         vectors[T_SYSCALL], DPL_USER);
3325     initlock(&tickslock, "time");
3326 }
```

# Initialize IDT

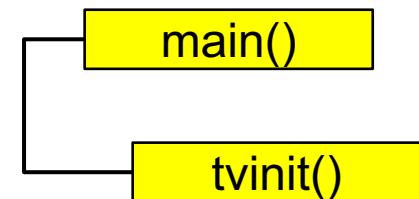


# 3316 void 3317 tvinit(void)

```
3318 {  
3319     int i;  
3320  
3321     for(i = 0; i < 256; i++)  
3322         SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);  
3323     SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3,  
3324  
3325         vectors[T_SYSCALL], DPL_USER);  
3326 }
```

## Initialize IDT

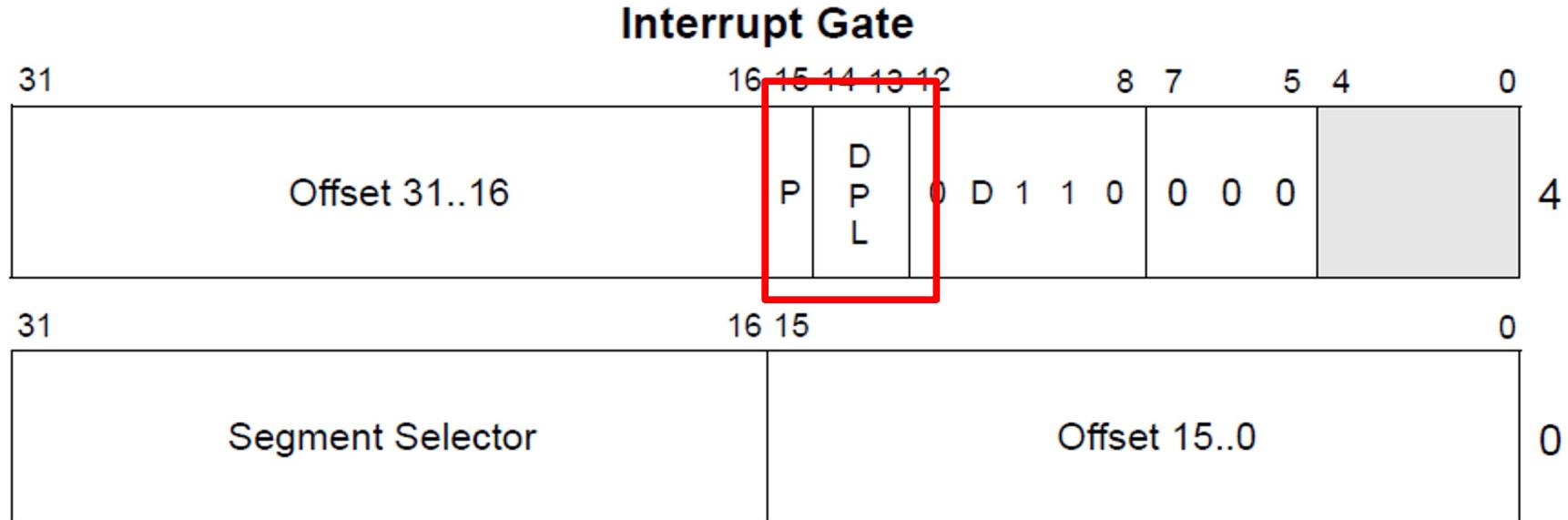
- System call interrupt vector (T\_SYSCALL)



# Protection

- Generally user code cannot invoke `int 0x..`
  - i.e., can't issue `int 14` (a page fault)
- OS configures the IDT in such a manner that invocation of all `int X` instructions besides `0x40` triggers a general protection fault exception
  - Interrupt vector 13

# Remember this slide: Interrupt descriptor (an entry in the IDT)

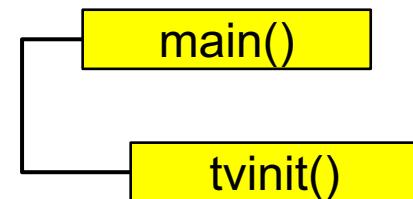


- Interrupt is allowed if...
  - current privilege level (CPL) is less or equal to descriptor privilege level (DPL)
- User cannot invoke `int 0x32`

```
3316 void
3317 tvinit(void)
3318 {
3319     int i;
3320
3321     for(i = 0; i < 256; i++)
3322         SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
3323     SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3,
3324                                         vectors[T_SYSCALL], DPL_USER);
3325     initlock(&tickslock, "time");
3326 }
```

# Initialize IDT

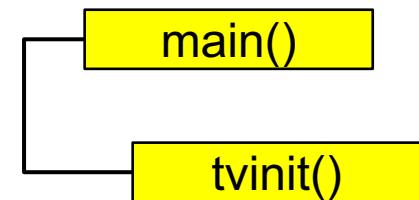
- A couple of important details



```
3316 void  
3317 tvinit(void)  
3318 {  
3319     int i;  
3320  
3321     for(i = 0; i < 256; i++)  
3322         SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);  
3323     SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3,  
3324  
3325         vectors[T_SYSCALL], DPL_USER);  
3326 }
```

# Initialize IDT

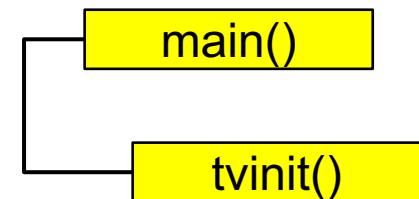
- Only T\_SYSCALL can be invoked from user level



```
3316 void  
3317 tvinit(void)  
3318 {  
3319     int i;  
3320  
3321     for(i = 0; i < 256; i++)  
3322         SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);  
3323     SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3,  
3324  
3325         vectors[T_SYSCALL], DPL_USER);  
3326 }
```

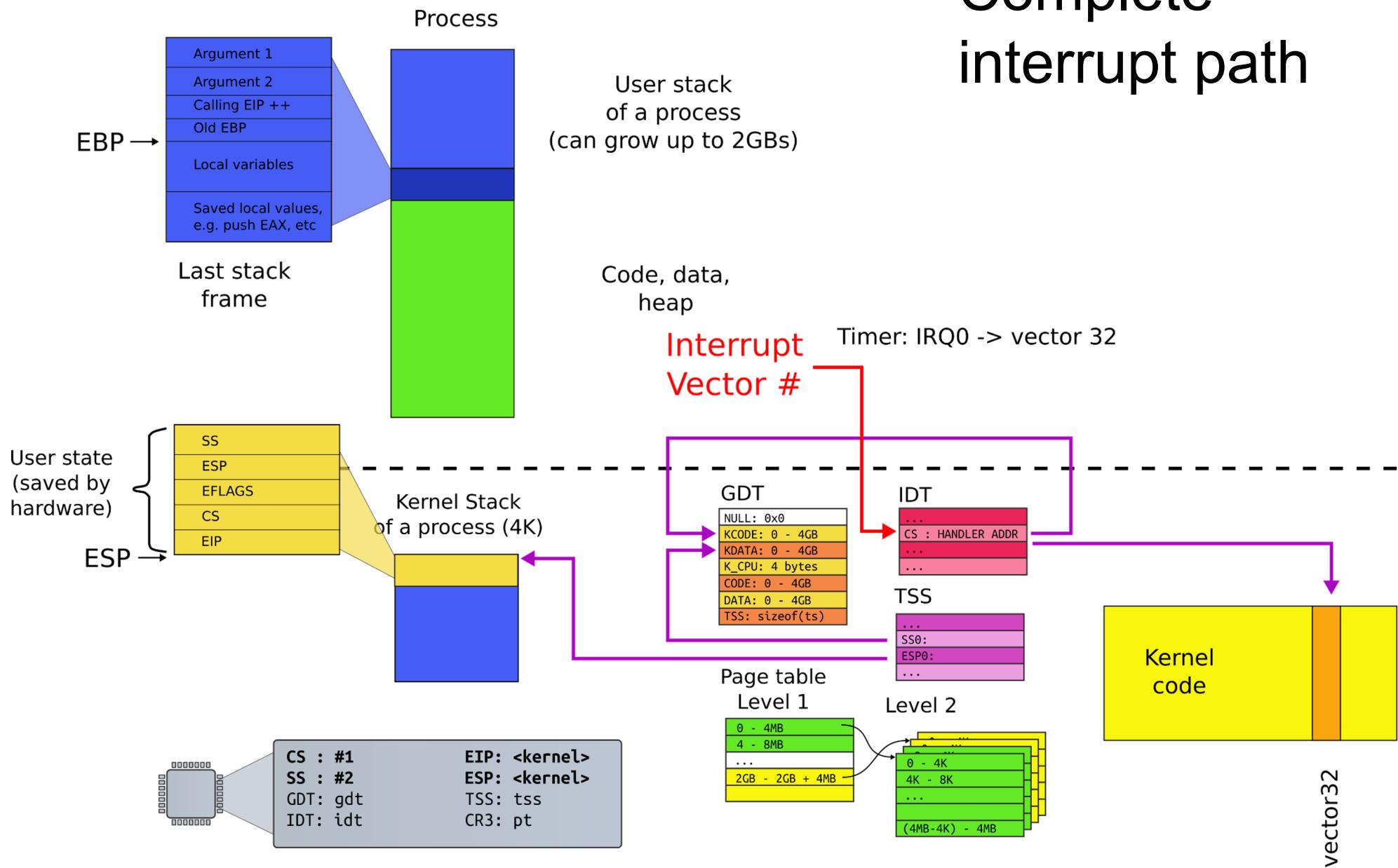
# Initialize IDT

- Syscall is a “trap”
- i.e., does not disable interrupts



# Interrupt path through the xv6 kernel

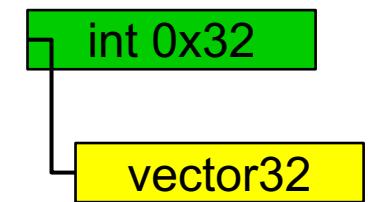
# Complete interrupt path



# Timer Interrupt (int 0x32)

vector32:

```
pushl $0 // error code
```

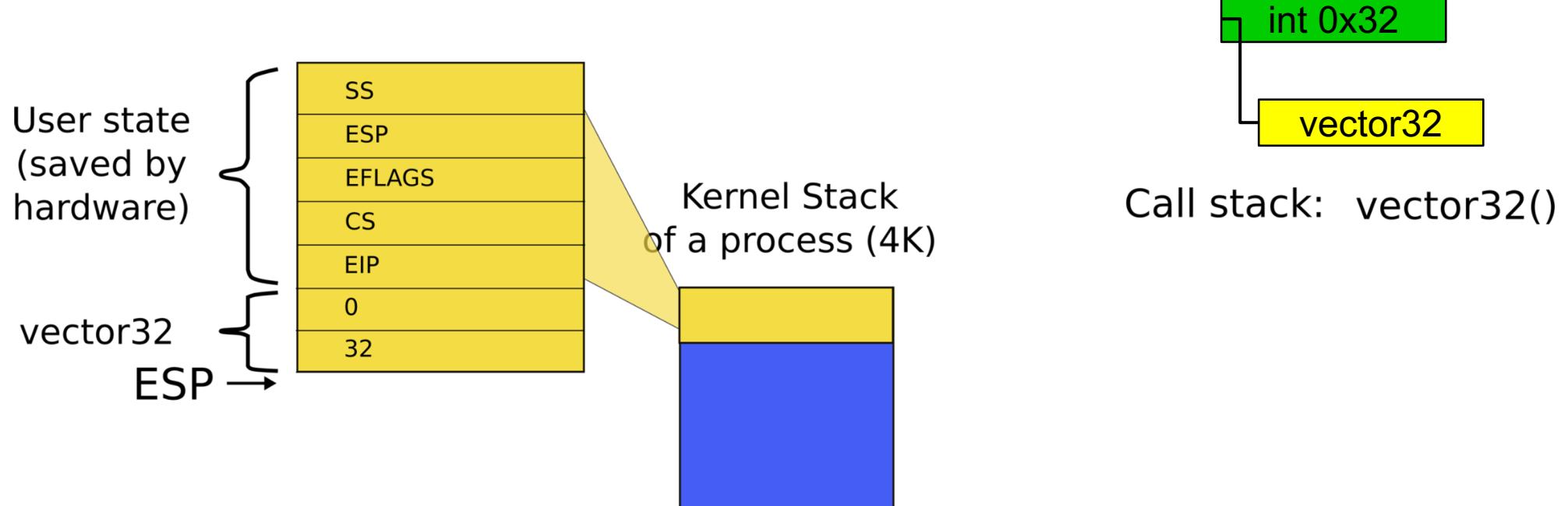


```
pushl $32 // vector #
```

```
jmp alltraps
```

- Automatically generated
- From vectors.pl
- vector.S

# Kernel stack after interrupt



3254 alltraps:

3255 # Build trap frame.

3256 pushl %ds

3257 pushl %es

3258 pushl %fs

3259 pushl %gs

3260 **pushal**

3261

3262 # Set up data segments.

3263 movw \$(SEG\_KDATA<<3), %ax

3264 movw %ax, %ds

3265 movw %ax, %es

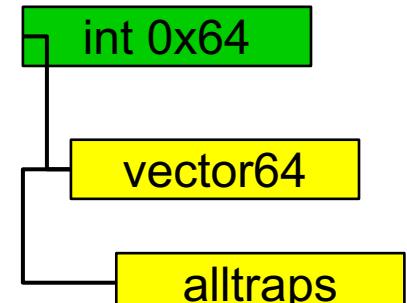
3266

3267 # Call trap(tf), where tf=%esp

3268 pushl %esp

3269 call trap

# alltraps()



# pusha

.An assembler instruction that saves all registers on the stack

.[https://c9x.me/x86/html/file\\_module\\_x86\\_id\\_270.html](https://c9x.me/x86/html/file_module_x86_id_270.html)

Temporary = ESP;

Push(EAX);

Push(ECX);

Push(EDX);

Push(EBX);

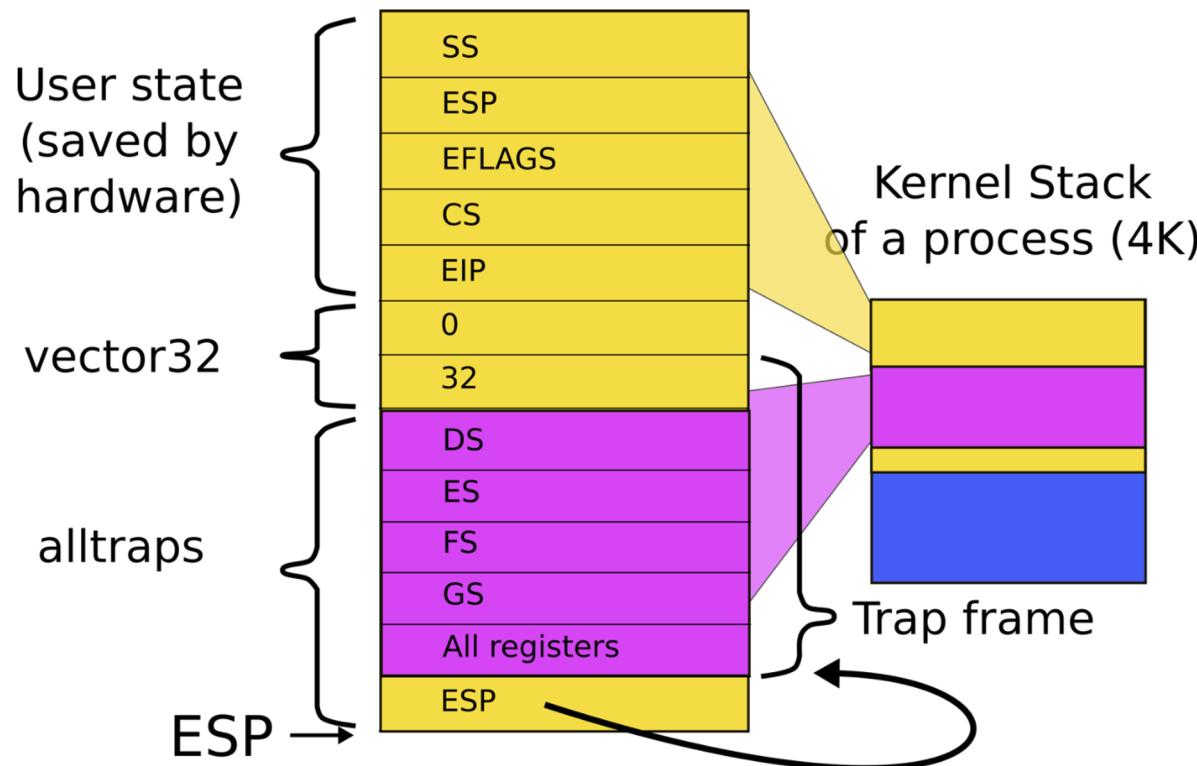
Push(Temporary);

Push(EBP);

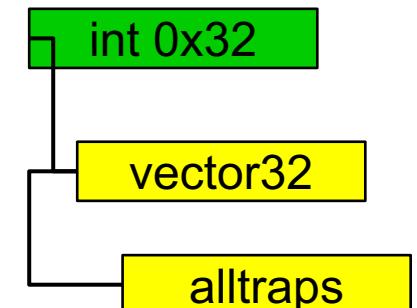
Push(ESI);

Push(EDI);

# Kernel stack after interrupt

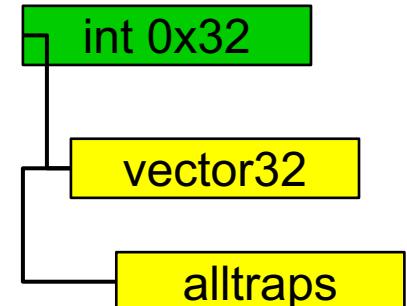


Call stack: `vector32()`  
`alltraps()`



# The end result: call trap()

```
3254 alltraps:  
3255 # Build trap frame.  
3256 pushl %ds  
3257 pushl %es  
3258 pushl %fs  
3259 pushl %gs  
3260 pushal  
3261  
3262 # Set up data and per-cpu segments.  
3263 movw $(SEG_KDATA<<3), %ax  
3264 movw %ax, %ds  
3265 movw %ax, %es  
3266 movw $(SEG_KCPU<<3), %ax  
3267 movw %ax, %fs  
3268 movw %ax, %gs  
3269  
3270 # Call trap(tf), where tf=%esp  
3271 pushl %esp  
3272 call trap
```

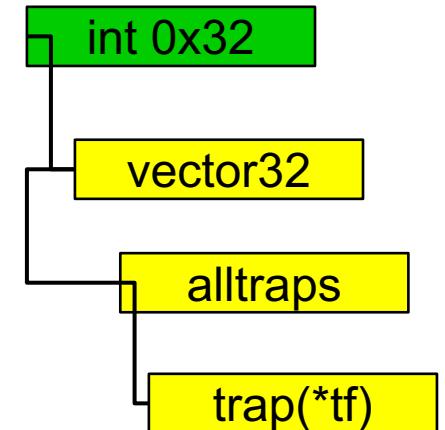


```

3351 trap(struct trapframe *tf)
3352 {
...
3363     switch(tf->trapno){
3364     case T_IRQ0 + IRQ_TIMER:
3365         if(cpu->id == 0){
3366             acquire(&tickslock);
3367             ticks++;
3368             wakeup(&ticks);
3369             release(&tickslock);
3370         }
3371         break;
...
3423     if(proc && proc->state == RUNNING
        && tf->trapno == T_IRQ0+IRQ_TIMER)
3424         yield();

```

All interrupts, e.g.  
 timer interrupt end  
 up in a single  
 function: trap()



3004 alltraps:

...

3020 # Call trap(tf), where tf=%esp

3021 pushl %esp

3022 call trap

3023 addl \$4, %esp

3024

3025 # Return falls through to trapret...

3026 .globl trapret

3027 trapret:

3028 popal

3029 popl %gs

3030 popl %fs

3031 popl %es

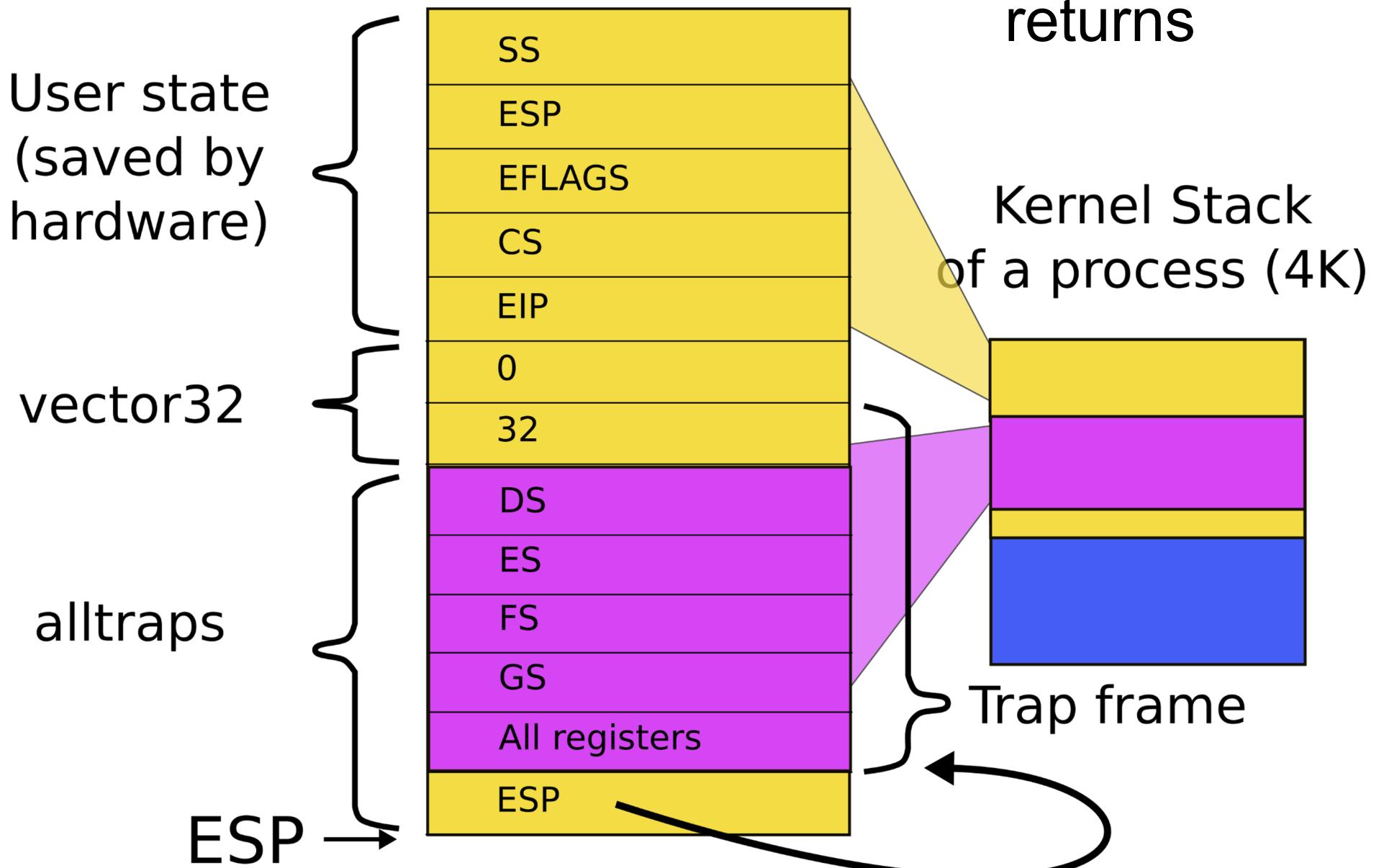
3032 popl %ds

3033 addl \$0x8, %esp # trapno and errcode

3034 iret

# alltraps(): exit from the interrupt

## Stack after trap() returns



3004 alltraps:

...

3020 # Call trap(tf), where tf=%esp

3021 pushl %esp

3022 call trap

3023 addl \$4, %esp

3024

3025 # Return falls through to trapret...

3026 .globl trapret

3027 trapret:

3028 popal

3029 popl %gs

3030 popl %fs

3031 popl %es

3032 popl %ds

3033 addl \$0x8, %esp # trapno and errcode

3034 iret

# alltraps(): exiting

.Restore all  
registers

.Exit into user  
.iret

# Return from an interrupt

.Starts with IRET

1.Restore the CS and EIP registers to their values prior to the interrupt or exception

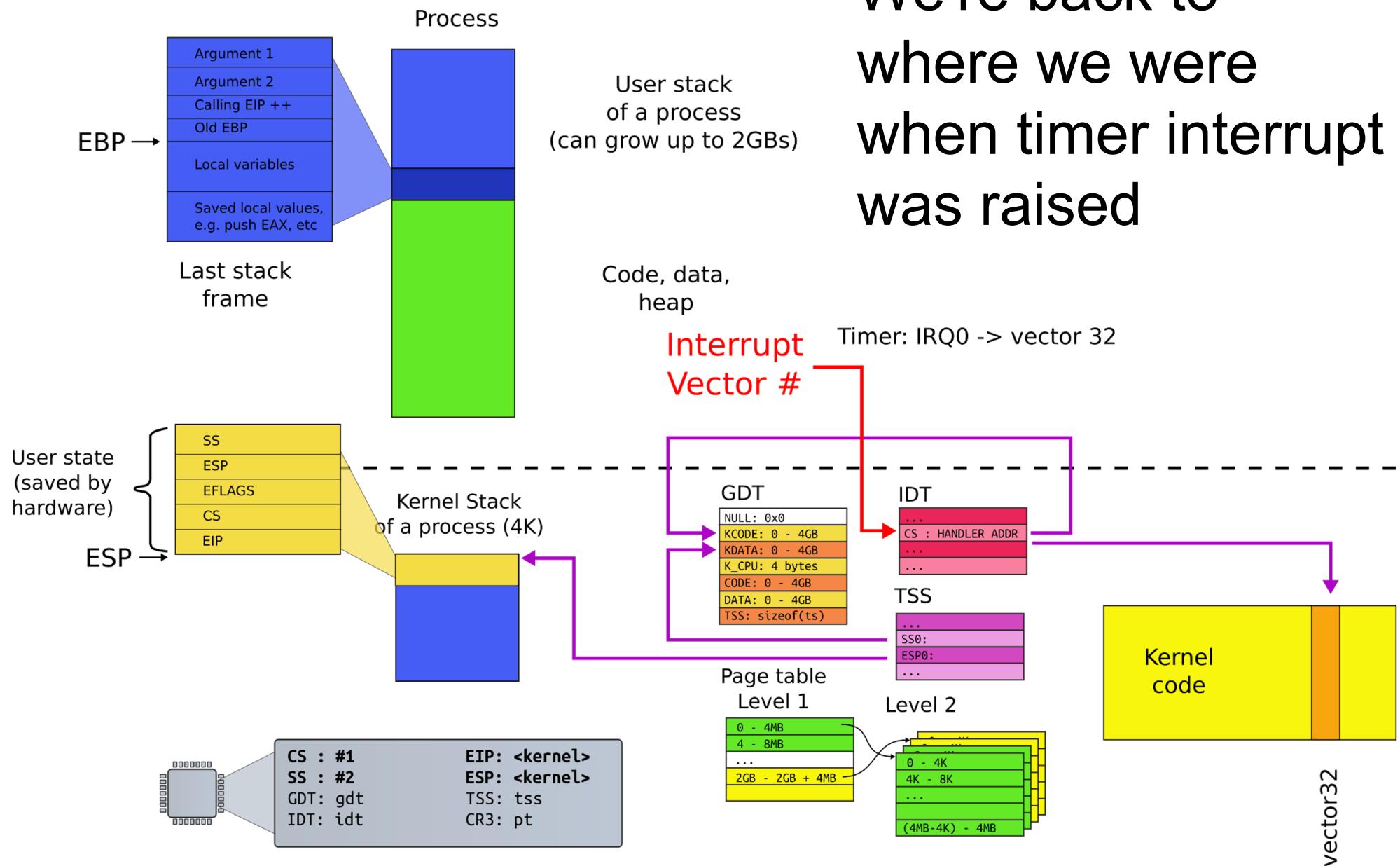
2.Restore EFLAGS

3.Restore SS and ESP to their values prior to interrupt

-This results in a stack switch

4.Resume execution of interrupted procedure

We're back to where we were when timer interrupt was raised



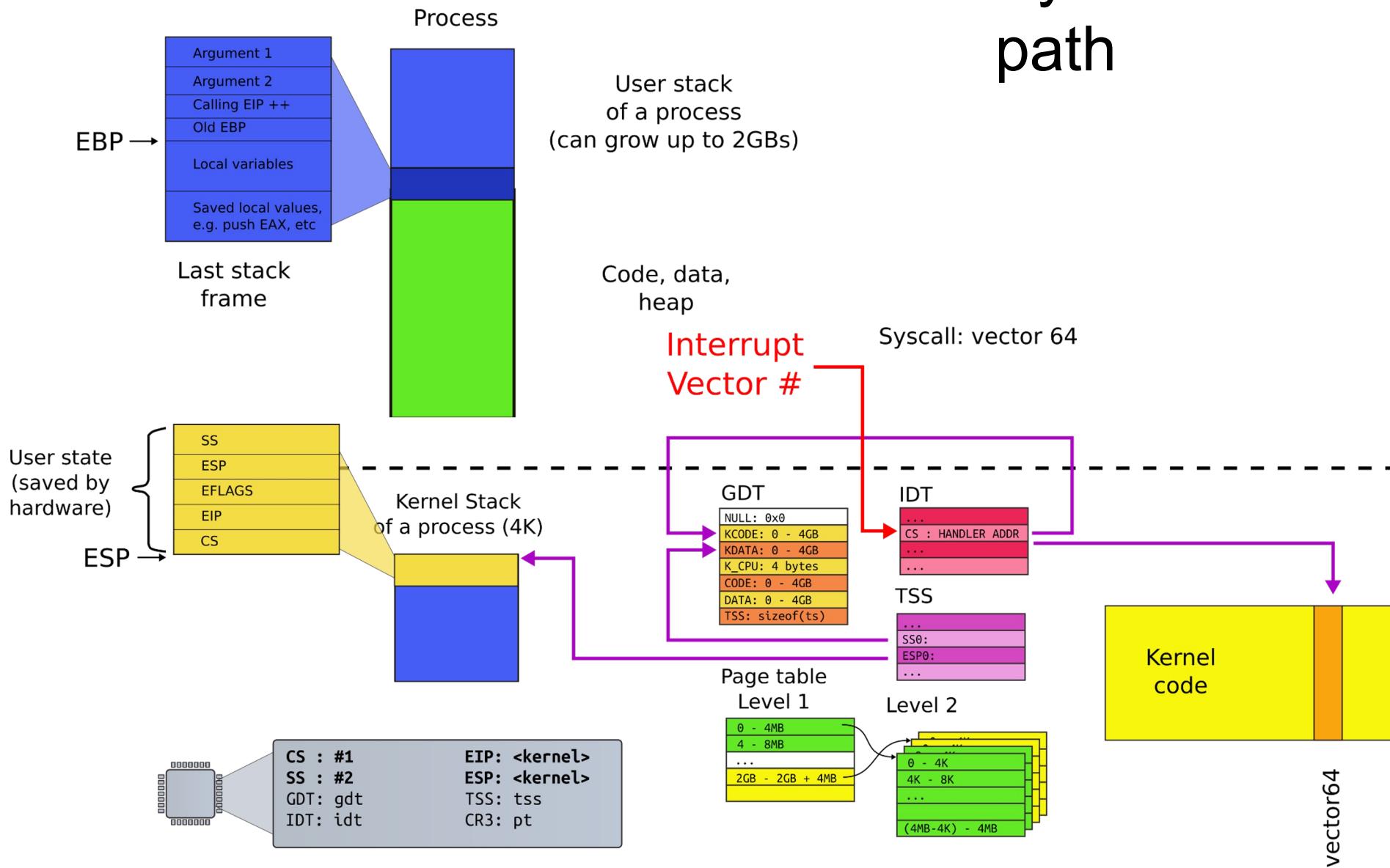
# System Calls

(**int 0x40**)

# Software interrupts can be used to implement system calls

- The int N instruction provides a secure mechanism for kernel invocation
  - i.e., user can enter the kernel
  - But through a well-defined entry point
    - System call handler
  - Xv6 uses vector 0x40 (or 64)
  - You can choose any other unused vector
  - Linux uses 0x80
    - Well now it uses sysenter instead of int 0x80 as it is faster

# System call path



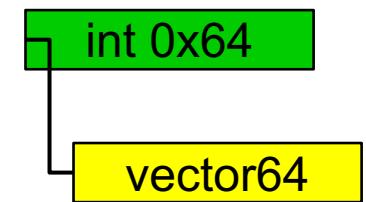
# Where does IDT (entry 64) point to?

vector64:

```
pushl $0 // error code
```

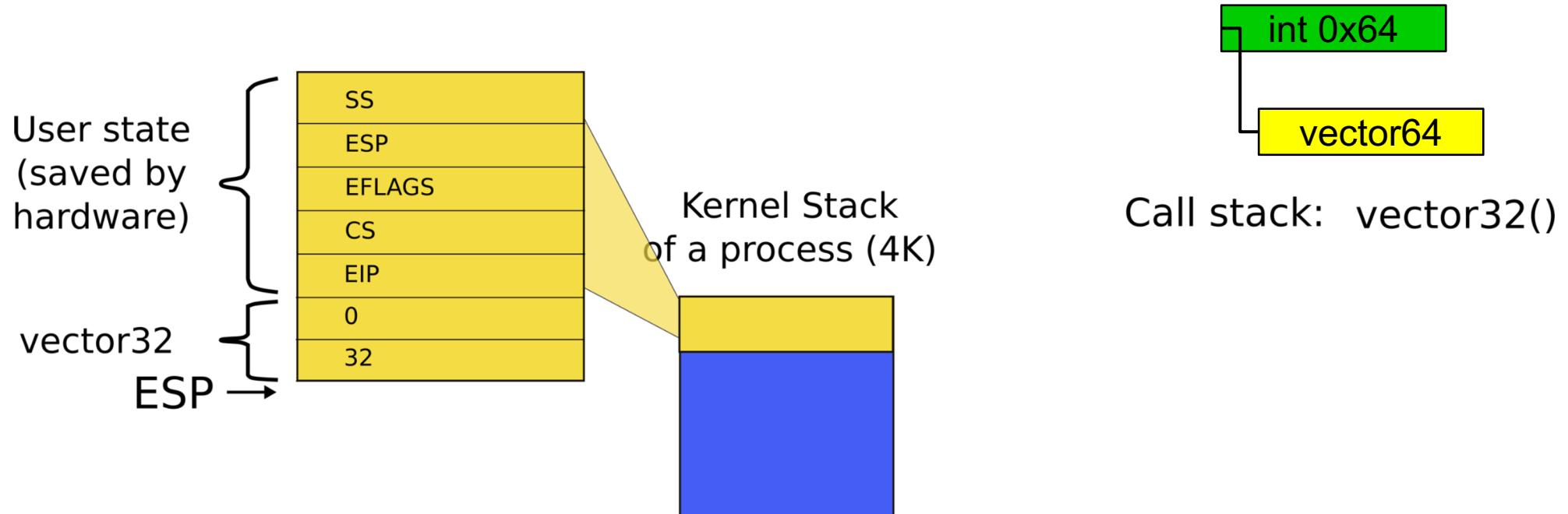
```
pushl $64 // vector #
```

```
jmp alltraps
```



- Automatically generated
- From vectors.pl
- vector.S

# Kernel stack inside system call



3254 alltraps:

3255 # Build trap frame.

3256 pushl %ds

3257 pushl %es

3258 pushl %fs

3259 pushl %gs

3260 **pushal**

3261

3262 # Set up data and per-cpu segments.

3263 movw \$(SEG\_KDATA<<3), %ax

3264 movw %ax, %ds

3265 movw %ax, %es

3266 movw \$(SEG\_KCPU<<3), %ax

3267 movw %ax, %fs

3268 movw %ax, %gs

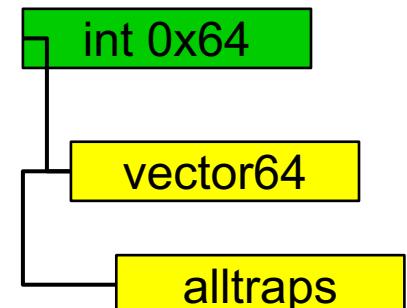
3269

3270 # Call trap(tf), where tf=%esp

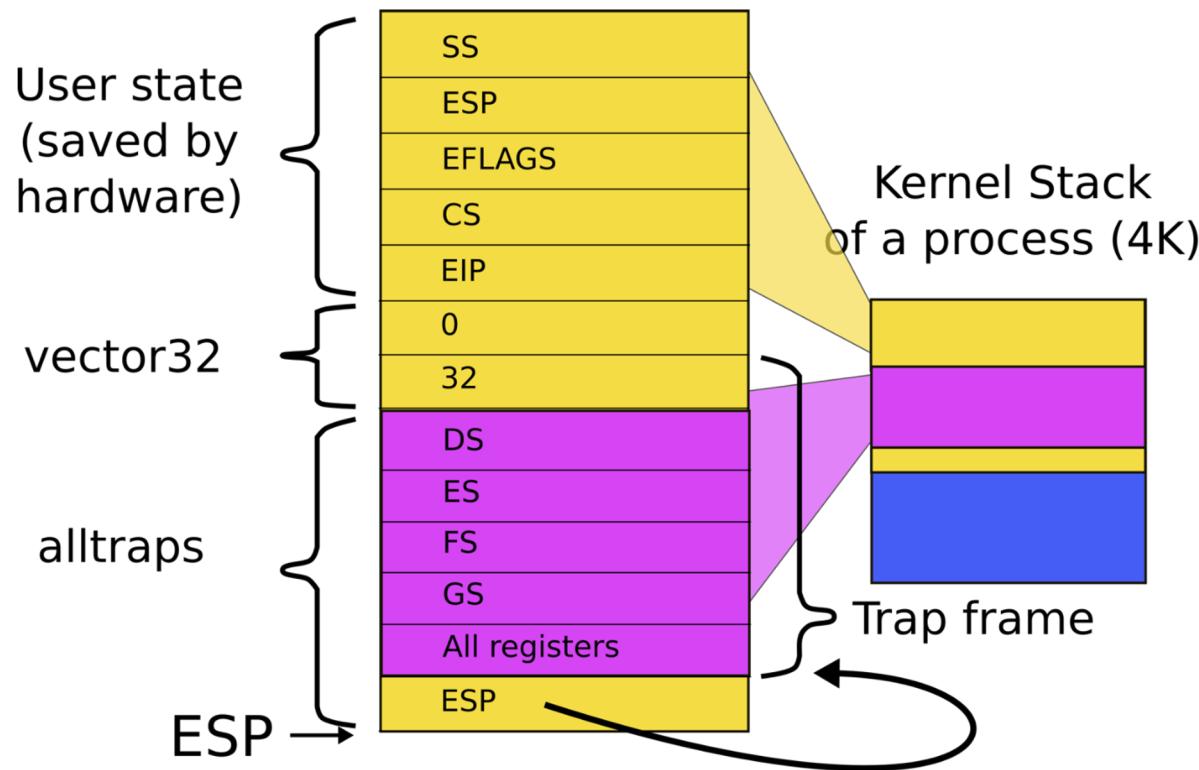
3271 pushl %esp

3272 call trap

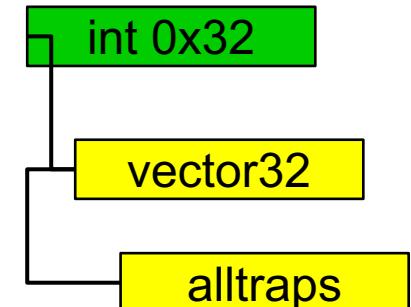
# alltraps()



# Kernel stack inside system call

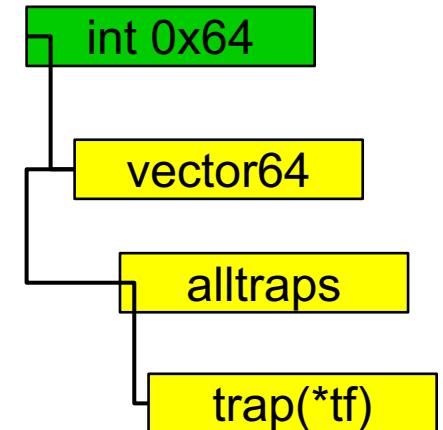


Call stack: `vector32()`  
`alltraps()`



```
3351 trap(struct trapframe *tf)
3352 {
3353     if(tf->trapno == T_SYSCALL){
3354         if(proc->killed)
3355             exit();
3356         proc->tf = tf;
3357         syscall();
3358         if(proc->killed)
3359             exit();
3360     return;
3361 }
3362
3363 switch(tf->trapno){
3364 case T_IRQ0 + IRQ_TIMER:
```

## System call handling inside trap()

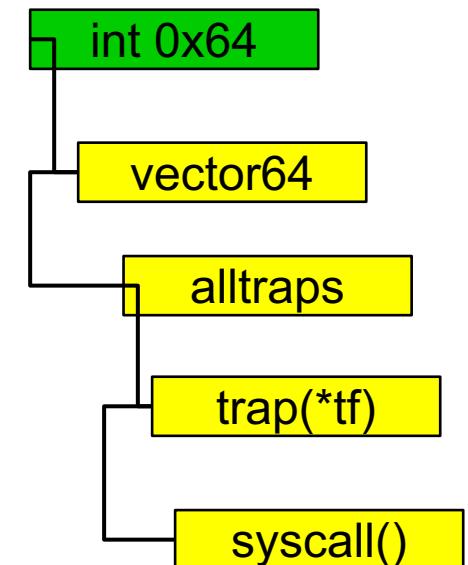


# Syscall number

- .System call number is passed in the %eax register
- .To distinguish which syscall to invoke,
  - e.g., `sys_read`, `sys_exec`, etc.
- .`alltrap()` saves it along with all other registers

# syscall(): get the number from the trap frame

```
3625 syscall(void)
3626 {
3627     int num;
3628
3629     num = proc->tf->eax;
3630     if(num > 0 && num < NELEM(syscalls) && syscalls[num])    {
3631         proc->tf->eax = syscalls[num]();
3632     } else {
3633         cprintf("%d %s: unknown sys call %d\n",
3634             proc->pid, proc->name, num);
3635         proc->tf->eax = -1;
3636     }
3637 }
```



# syscall(): process a syscall from the table

```
3625 syscall(void)
3626 {
3627     int num;
3628
3629     num = proc->tf->eax;
3630     if(num > 0 && num < NELEM(syscalls) && syscalls[num])      {
3631         proc->tf->eax = syscalls[num]();
3632     } else {
3633         cprintf("%d %s: unknown sys call %d\n",
3634             proc->pid, proc->name, num);
3635         proc->tf->eax = -1;
3636     }
3637 }
```

```
3600 static int (*syscalls[])(void) = {  
3601 [SYS_fork] sys_fork,  
3602 [SYS_exit] sys_exit,  
3603 [SYS_wait] sys_wait,  
3604 [SYS_pipe] sys_pipe,  
3605 [SYS_read] sys_read,  
3606 [SYS_kill] sys_kill,  
3607 [SYS_exec] sys_exec,  
3608 [SYS_fstat] sys_fstat,  
3609 [SYS_chdir] sys_chdir,  
3610 [SYS_dup] sys_dup,  
3611 [SYS_getpid] sys_getpid,  
3612 [SYS_sbrk] sys_sbrk,  
3613 [SYS_sleep] sys_sleep,  
3614 [SYS_uptime] sys_uptime,  
3615 [SYS_open] sys_open,  
3616 [SYS_write] sys_write,  
3617 [SYS_mknod] sys_mknod,  
3618 [SYS_unlink] sys_unlink,  
3619 [SYS_link] sys_link,  
3620 [SYS_mkdir] sys_mkdir,  
3621 [SYS_close] sys_close,  
3622 };
```

# System call table

# How do user programs access system calls?

- It would be weird to write

8410 pushl \$argv

8411 pushl \$init

8412 pushl \$0 // where caller pc would be

8413 movl \$SYS\_exec, %eax

8414 int \$T\_SYSCALL

.... every time we want to invoke a system call

- This is an example for the exec() system call

```
// system calls  
  
int fork(void);  
  
int exit(void) __attribute__((noreturn));  
  
int wait(void);  
  
int pipe(int*);  
  
int write(int, void*, int);  
  
int read(int, void*, int);  
  
int close(int);  
  
int kill(int);  
  
int exec(char*, char**);  
  
int open(char*, int);  
  
int mknod(char*, short, short);  
  
int unlink(char*);  
  
int fstat(int fd, struct stat*);  
  
int link(char*, char*);  
  
...
```

## user.h

- .user.h defines system call prototypes
- .Compiler can generate correct system call stacks
- .Remember calling conventions?
- .Arguments on the stack

# Example

• From cat.asm

• if (write(1, buf, n) != n)

A3: 53                push ebx

a4: 68 00 0b 00 00    push 0xb00

a9: 6a 01            push 0x1

ab: e8 c2 02 00 00    call 372 <write>

- Note, different versions of gcc
- and different optimization levels
- Will generate slightly different code

# Example

• From cat.asm

• if (write(1, buf, n) != n)

a0: 89 5c 24 08        mov %ebx,0x8(%esp)

a4: c7 44 24 04 00 0b 00    movl \$0xb00,0x4(%esp)

ab: 00

ac: c7 04 24 01 00 00 00    movl \$0x1,(%esp)

b3: e8 aa 02 00 00        call 362 <write>

# Example

- From cat.asm

- if (write(1, buf, n) != n)

a0: 89 5c 24 08        mov %ebx,0x8(%esp)

a4: c7 44 24 04 00 0b 00    movl \$0xb00,0x4(%esp)

ab: 00

ac: c7 04 24 01 00 00 00    movl \$0x1,(%esp)

b3: e8 aa 02 00 00        call 362 <write>

# Example

• From cat.asm

• if (write(1, buf, n) != n)

a0: 89 5c 24 08      mov %ebx,0x8(%esp)

a4: c7 44 24 04 00 0b 00    movl \$0xb00,0x4(%esp)

ab: 00

ac: c7 04 24 01 00 00 00    movl \$0x1,(%esp)

b3: e8 aa 02 00 00      call 362 <write>

- .Still not clear...
- .The header file allows compiler to generate a call side invocation,
  - e.g., push arguments on the stack
- .But where is the system call invocation itself
  - e.g., `int $T_SYSCALL`

```
8450 #include "syscall.h"
8451 #include "traps.h"
8452
8453 #define SYSCALL(name) \
8454 .globl name; \
8455 name: \
8456     movl $SYS_## name, %eax; \
8457     int $T_SYSCALL; \
8458     ret
8459
8460 SYSCALL(fork)
8461 SYSCALL(exit)
8462 SYSCALL(wait)
8463 SYSCALL(pipe)
8464 SYSCALL(read)
```

usys.S

.Xv6 uses a SYSCALL macro to define a function for each system call invocation  
.E.g., fork() to invoke the “fork” system call

# Example

- Write system call from cat.asm

00000362 <write>:

SYSCALL(write)

362: b8 10 00 00 00        mov \$0x10,%eax

367: cd 40                int \$0x40

369: c3                    ret

# System call arguments

- Where are the system call arguments?
- How does kernel access them?
- And returns results?

# Example

- Write system call

- if (write(1, buf, n) != n)

5876 int

5877 sys\_write(void)

5878 {

5879 struct file \*f;

5880 int n;

5881 char \*p;

5882

5883 if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)

5884 return -1;

5885 return filewrite(f, p, n);

5886 }

# Example

- Write system call

- if (write(1, buf, n) != n)

5876 int

5877 sys\_write(void)

5878 {

5879 struct file \*f;

5880 int n;

5881 char \*p;

5882

5883 if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)

5884 return -1;

5885 return filewrite(f, p, n);

5886 }

3543 // Fetch the nth 32-bit system call argument.

3544 int

3545 argint(int n, int \*ip)

3546 {

3547 return fetchint(proc->tf->esp + 4 + 4\*n, ip);

3548 }

3515 // Fetch the int at addr from the current process.

3516 int

3517 fetchint(uint addr, int \*ip)

3518 {

3519 if(addr >= proc->sz || addr+4 > proc->sz)

3520 return -1;

3521 \*ip = \*(int\*)(addr);

3522 return 0;

3523 }

**argint(int n, int \*ip)**

3543 // Fetch the nth 32-bit system call argument.

3544 int

3545 argint(int n, int \*ip)

3546 {

3547   return fetchint(proc->tf->esp + 4 + 4\*n, ip);

3548 }

3515 // Fetch the int at addr from the current process.

3516 int

3517 fetchint(uint addr, int \*ip)

3518 {

3519   if(addr >= proc->sz || addr+4 > proc->sz)

3520       return -1;

3521   \*ip = \*(int\*)(addr);

3522   return 0;

3523 }

**argint(int n, int \*ip)**

3543 // Fetch the nth 32-bit system call argument.

```
3544 int  
3545 argint(int n, int *ip)  
3546 {  
3547     return fetchint(proc->tf->esp + 4 + 4*n, ip);  
3548 }
```

.Start with the address where current user stack is (esp)

3515 // Fetch the int at addr from the current process.

```
3516 int  
3517 fetchint(uint addr, int *ip)  
3518 {  
3519     if(addr >= proc->sz || addr+4 > proc->sz)  
3520         return -1;  
3521     *ip = *(int*)(addr);  
3522     return 0;  
3523 }
```

**argint(int n, int \*ip)**

3543 // Fetch the nth 32-bit system call argument.

3544 int

3545 argint(int n, int \*ip)

3546 {

3547   return fetchint(proc->tf->esp + 4 + 4\*n, ip);

3548 }

.Skip return eip

3515 // Fetch the int at addr from the current process.

3516 int

3517 fetchint(uint addr, int \*ip)

3518 {

3519   if(addr >= proc->sz || addr+4 > proc->sz)

3520       return -1;

3521   \*ip = \*(int\*)(addr);

3522   return 0;

3523 }

argint(int n, int \*ip)

3543 // Fetch the nth 32-bit system call argument.

3544 int

3545 argint(int n, int \*ip)

3546 {

3547   return fetchint(proc->tf->esp + 4 + 4\*n, ip);

3548 }

.Fetch n'th  
argument



3515 // Fetch the int at addr from the current process.

3516 int

3517 fetchint(uint addr, int \*ip)

3518 {

3519   if(addr >= proc->sz || addr+4 > proc->sz)

3520       return -1;

3521       \*ip = \*(int\*)(addr);

3522       return 0;

3523 }

argint(int n, int \*ip)

3543 // Fetch the nth 32-bit system call argument.

3544 int

3545 argint(int n, int \*ip)

3546 {

3547 return fetchint(proc->tf->esp + 4 + 4\*n, ip);

3548 }

3515 // Fetch the int at addr from the current process.

3516 int

3517 fetchint(uint addr, int \*ip)

3518 {

3519 if(addr >= proc->sz || addr+4 > proc->sz)

3520 return -1;

3521 \*ip = \*(int\*)(addr);

3522 return 0;

3523 }

**fetchint(uint addr, int \*ip)**

3543 // Fetch the nth 32-bit system call argument.

3544 int

3545 argint(int n, int \*ip)

3546 {

3547 return fetchint(proc->tf->esp + 4 + 4\*n, ip);

3548 }

3515 // Fetch the int at addr from the current process.

3516 int

3517 fetchint(uint addr, int \*ip)

3518 {

3519 if(addr >= proc->sz || addr+4 > proc->sz)

3520 return -1;

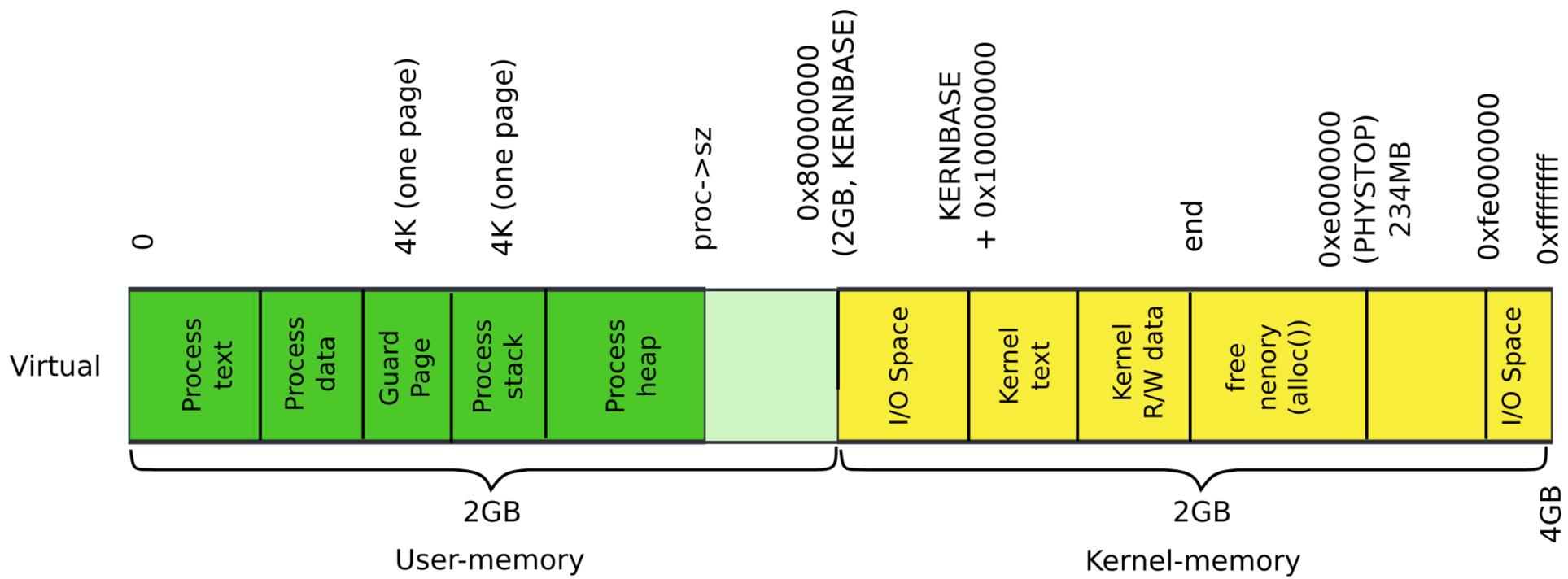
3521 \*ip = \*(int\*)(addr);

3522 return 0;

3523 }

**fetchint(uint addr, int \*ip)**

# Process address space



# Any idea for what argptr() shall do?

- Write system call

- if (write(1, buf, n) != n)

5876 int

5877 sys\_write(void)

5878 {

5879 struct file \*f;

5880 int n;

5881 char \*p;

5882

5883 if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)

5884 return -1;

5885 return filewrite(f, p, n);

5886 }

- Remember, buf is a pointer to a region of memory

- i.e., a buffer

- of size n

```
3550 // Fetch the nth word-sized system call argument as a pointer  
3551 // to a block of memory of size n bytes. Check that the pointer  
3552 // lies within the process address space.
```

```
3553 int
```

```
3554 argptr(int n, char **pp, int size)
```

```
3555 {
```

```
3556     int i;
```

```
3557
```

```
3558     if(argint(n, &i) < 0)
```

```
3559         return -1;
```

```
3560     if((uint)i >= proc->sz || (uint)i+size > proc->sz)
```

```
3561         return -1;
```

```
3562     *pp = (char*)i;
```

```
3563     return 0;
```

```
3564 }
```

.Check that the pointer  
to the buffer is sound

**argptr(uint addr, int \*ip)**

```
3550 // Fetch the nth word-sized system call argument as a pointer  
3551 // to a block of memory of size n bytes. Check that the pointer  
3552 // lies within the process address space.
```

```
3553 int
```

```
3554 argptr(int n, char **pp, int size)
```

```
3555 {
```

```
3556     int i;
```

```
3557
```

```
3558     if(argint(n, &i) < 0)
```

```
3559         return -1;
```

```
3560     if((uint)i >= proc->sz || (uint)i+size > proc->sz)
```

```
3561         return -1;
```

```
3562     *pp = (char*)i;
```

```
3563     return 0;
```

```
3564 }
```

.Check that the buffer  
is in user memory

**argptr(uint addr, int \*ip)**

# Summary

- We've learned how system calls work

# Printing on the console

```
1317 main(void)
1318 {
1319     kinit1(end, P2V(4*1024*1024)); // phys page allocator
1320     kvmalloc(); // kernel page table
1321     mpinit(); // detect other processors
1322     lapicinit(); // interrupt controller
1323     seginit(); // segment descriptors
1324     cprintf("\ncpu%d: starting xv6\n\n", cpunum());
1325     picinit(); // another interrupt controller
1326     ioapicinit(); // another interrupt controller
1327     consoleinit(); // console hardware
1328     uartinit(); // serial port
1329     pinit(); // process table
1330     tvinit(); // trap vectors
1331     binit(); // buffer cache
1332     fileinit(); // file table
1333     ideinit(); // disk
1334     if(!ismp)
1335         timerinit(); // uniprocessor timer
1336     startothers(); // start other processors
1337     kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
1338     userinit(); // first user process
1339     mpmain(); // finish this processor's setup
1340 }
```

main()

```
8000 // Print to the console. only understands %d, %x, %p, %s.
```

```
8001 void
```

```
8002 cprintf(char *fmt, ...)
```

```
8003 {
```

```
...
```

```
8012 if (fmt == 0)
```

```
8013 panic("null fmt");
```

```
8014
```

```
8015 argp = (uint*)(void*)(&fmt + 1);
```

```
8016 for(i = 0; (c = fmt[i] & 0xff) != 0; i++){
```

```
8017 if(c != '%'){


```

```
8018 consputc(c);


```

```
8019 continue;


```

```
8020 }
```

```
8021 c = fmt[+i] & 0xff;
```

```
8022 if(c == 0)


```

```
8023 break;


```

```
8024 switch(c){


```

```
...
```

```
8032 case 's':
```

```
8033 if((s = (char*)*argp++) == 0)


```

```
8034 s = "(null)";


```

```
8035 for(; *s; s++)


```

```
8036 consputc(*s);


```

```
8037 break;


```

```
...
```

# Print on the screen

```
8150 void  
8151 consputc(int c)  
8152 {  
...  
8159 if(c == BACKSPACE){  
8160   uartputc('\b'); uartputc(' '); uartputc('\b');  
8161 } else  
8162   uartputc(c);  
8163 cgaputc(c);  
8164 }  
...  
8350 void  
8351 uartputc(int c)  
8352 {  
8353 int i;  
8354  
8355 if(!uart)  
8356   return;  
8357 for(i = 0; i < 128 && !(inb(COM1+5) & 0x20); i++)  
8358   microdelay(10);  
8359   outb(COM1+0, c);  
8360 }
```

# Print one character

```
1317 main(void)
1318 {
1319     kinit1(end, P2V(4*1024*1024)); // phys page allocator
1320     kvmalloc(); // kernel page table
1321     mpinit(); // detect other processors
1322     lapicinit(); // interrupt controller
1323     seginit(); // segment descriptors
1324     cprintf("\ncpu%d: starting xv6\n\n", cpunum());
1325     picinit(); // another interrupt controller
1326     ioapicinit(); // another interrupt controller
1327     consoleinit(); // console hardware
1328     uartinit(); // serial port
1329     pinit(); // process table
1330     tvinit(); // trap vectors
1331     binit(); // buffer cache
1332     fileinit(); // file table
1333     ideinit(); // disk
1334     if(!ismp)
1335         timerinit(); // uniprocessor timer
1336     startothers(); // start other processors
1337     kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
1338     userinit(); // first user process
1339     mpmain(); // finish this processor's setup
1340 }
```

main()

```
8000 // Print to the console. only understands %d, %x, %p, %s.
```

```
8001 void
```

```
8002 cprintf(char *fmt, ...)
```

```
8003 {
```

```
...
```

```
8012 if (fmt == 0)
```

```
8013 panic("null fmt");
```

```
8014
```

```
8015 argp = (uint*)(void*)(&fmt + 1);
```

```
8016 for(i = 0; (c = fmt[i] & 0xff) != 0; i++){
```

```
8017 if(c != '%'){


```

```
8018 consputc(c);


```

```
8019 continue;


```

```
8020 }
```

```
8021 c = fmt[+i] & 0xff;
```

```
8022 if(c == 0)


```

```
8023 break;


```

```
8024 switch(c){


```

```
...
```

```
8032 case 's':
```

```
8033 if((s = (char*)*argp++) == 0)


```

```
8034 s = "(null)";


```

```
8035 for(; *s; s++)


```

```
8036 consputc(*s);


```

```
8037 break;


```

```
...
```

# Print on the screen

```
8150 void  
8151 consputc(int c)  
8152 {  
...  
8159 if(c == BACKSPACE){  
8160     uartputc('\b'); uartputc(' '); uartputc('\b');  
8161 } else  
8162     uartputc(c);  
8163     cgaputc(c);  
8164 }  
...  
8350 void  
8351 uartputc(int c)  
8352 {  
8353     int i;  
8354  
8355     if(!uart)  
8356         return;  
8357     for(i = 0; i < 128 && !(inb(COM1+5) & 0x20); i++)  
8358         microdelay(10);  
8359     outb(COM1+0, c);  
8360 }
```

# Print one character (serial line)

```
8102 static ushort *crt = (ushort*)P2V(0xb8000); // CGA memory
8103
8104 static void
8105 cgaputc(int c)
8106 {
8107     int pos;
8108
8109 ...
8110
8111
8112
8113
8114
8115     if(c == '\n')
8116         pos += 80 - pos%80;
8117     else if(c == BACKSPACE){
8118         if(pos > 0) --pos;
8119     } else
8120         crt[pos++] = (c&0xff) | 0x0700; // black on white
8121
8122 ...
8123
8124
8125     if((pos/80) >= 24){ // Scroll up.
8126         memmove(crt, crt+80, sizeof(crt[0])*23*80);
8127         pos -= 80;
8128         memset(crt+pos, 0, sizeof(crt[0])*(24*80 - pos));
8129     }
8130
8131 ...
8132
8133
8134
8135
8136 }
```

# Print one character (display)

Thank you