CS238P: Operating Systems

Lecture 10: Interrupts and Exceptions

Anton Burtsev

February, 2018

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 (to avoid overflowing) and may switch to a process waiting on that packet)
    - Timer interrupt (OS may switch to another process)
- Finally, a third use-case
  - [It's also synchronous] For a while an interrupt, i.e., int 0xXX 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)

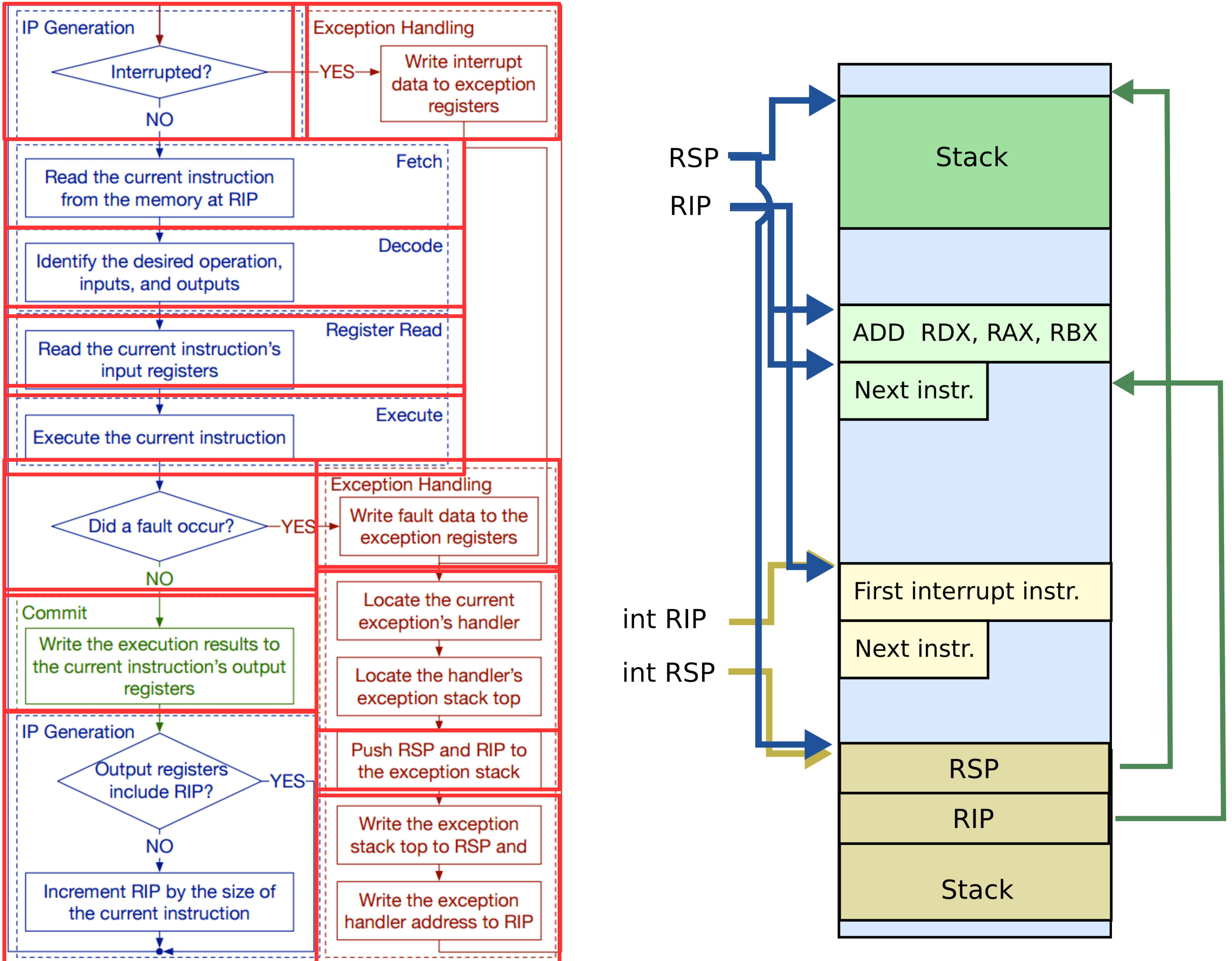
Synchronous

- Exceptions – react to an abnormal condition
  - E.g., page mapping (virtual address) is not present in the page table
    - Bring the swapped page back to memory (copy from disk)
    - Fix the page-table entry
  - Invoke a system call
    - Transition from user-level to kernel (more later)

Asynchronous

- Interrupts – preempt normal execution
  - Notify that something has happened
    - New packet from the network, disk I/O completed, timer tick, notification from another CPU)

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
Locate the current exception's handler
Locate the handler's exception stack top
Push RSP and RIP to the exception stack
Write the exception stack top to RSP and
Write the exception handler address to RIP
Commit
Write the execution results to the current instruction's output registers
IP Generation
Output registers include RIP?
YES
NO
Increment RIP by the size of the current instruction
RSP
RIP
int RIP
int RSP
Stack
ADD RDX, RAX, RBX
Next instr.
First interrupt instr.
Next instr.
RSP
RIP
Stack

Handling interrupts and exceptions

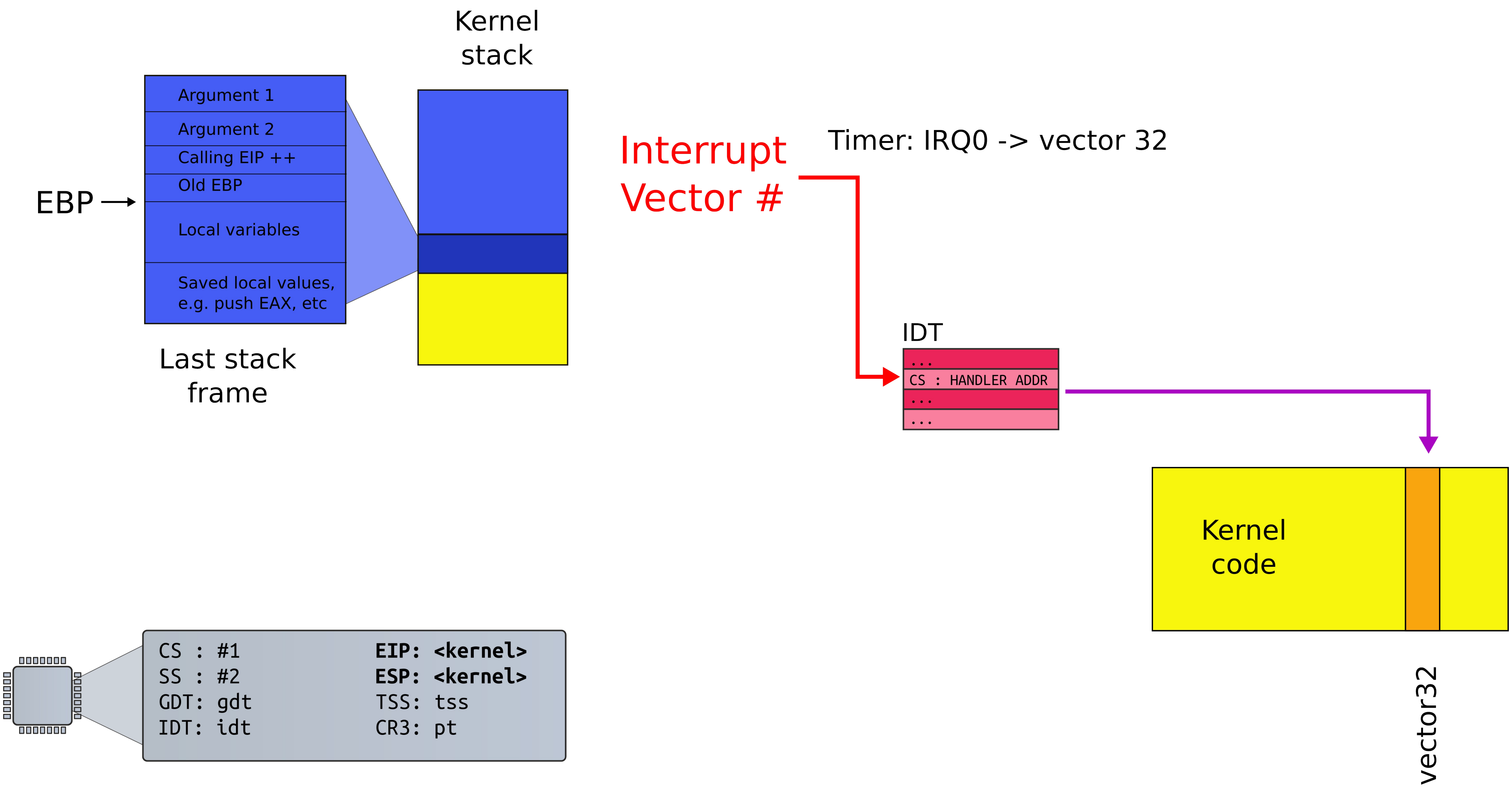
- In both synchronous and asynchronous cases the CPU follows the **same procedure**
  - Stop execution of the current program
  - Start execution of a handler
  - Processor accesses the handler through an entry in the Interrupt Descriptor Table (IDT)
- Each interrupt is defined by a number
  - E.g.,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

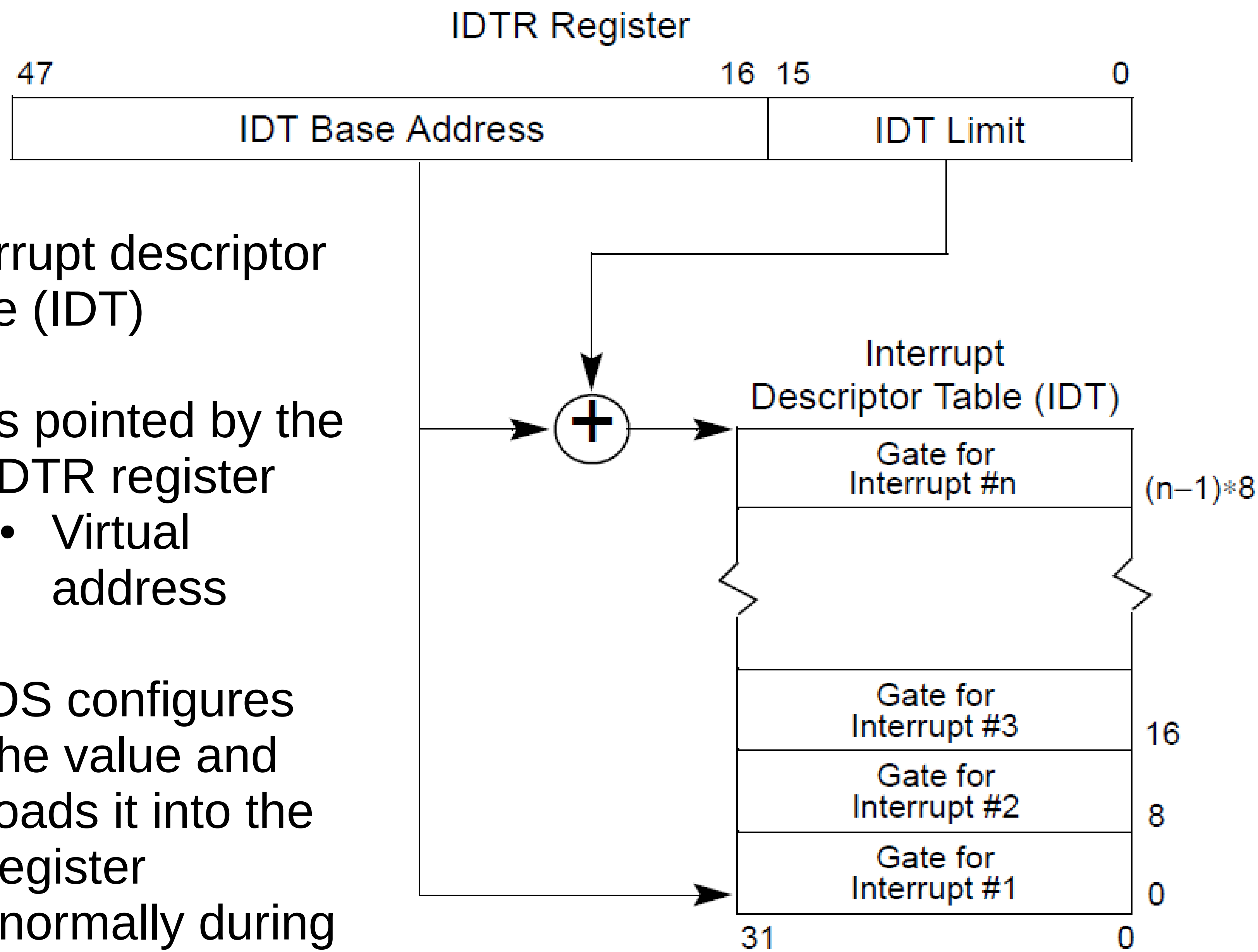
Case #1: Interrupt path no change in privilege level

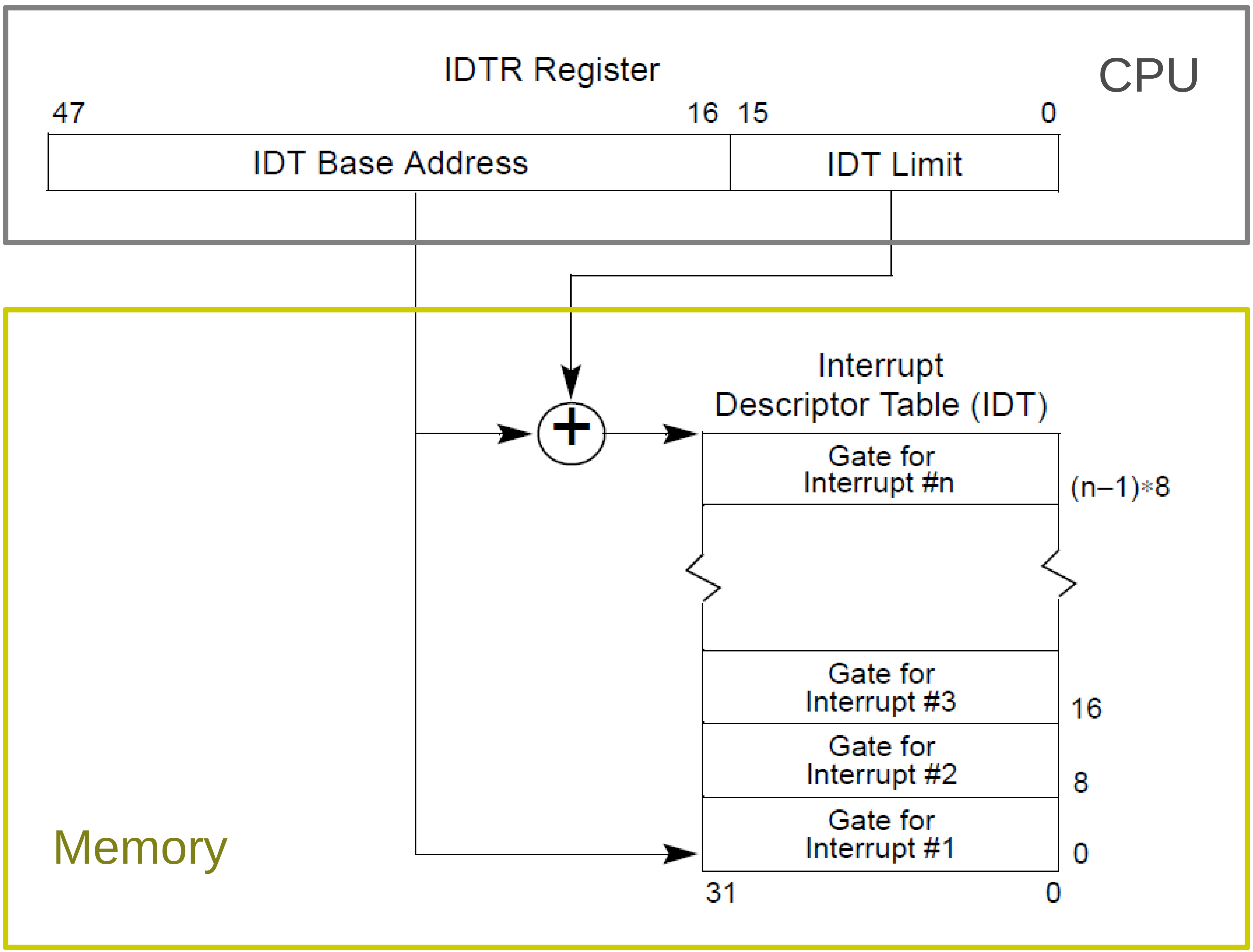
- e.g., we're already running in the kernel



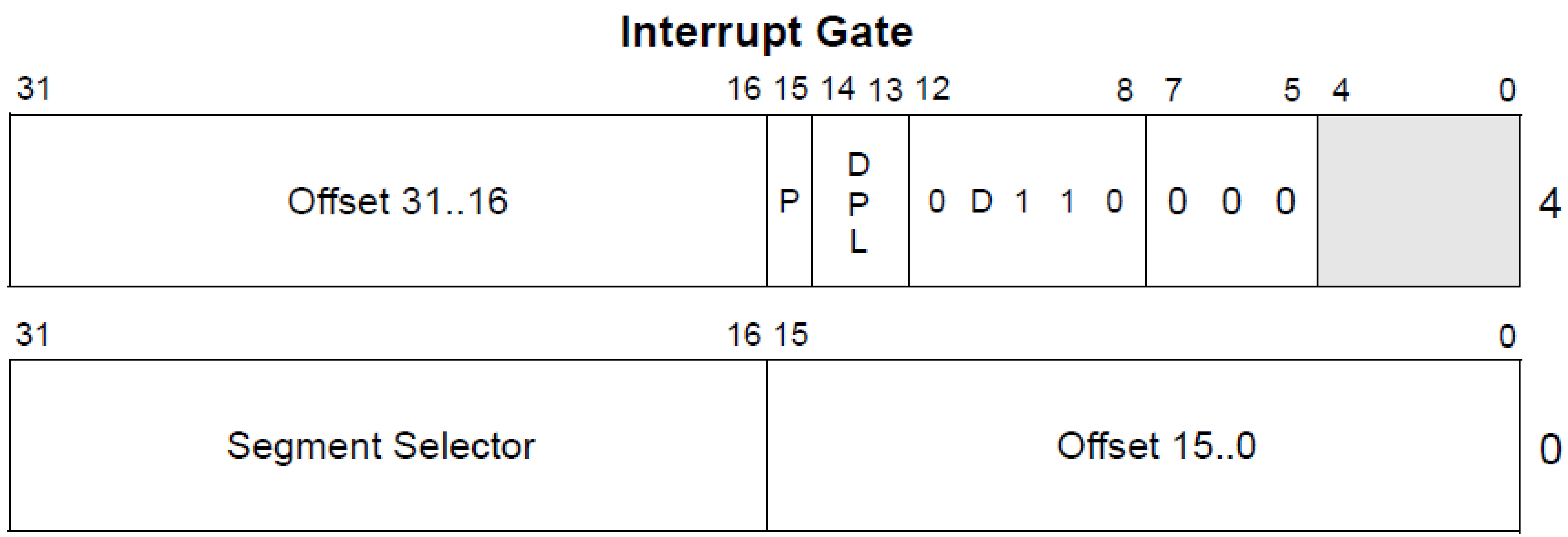
Interrupt descriptor table (IDT)

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

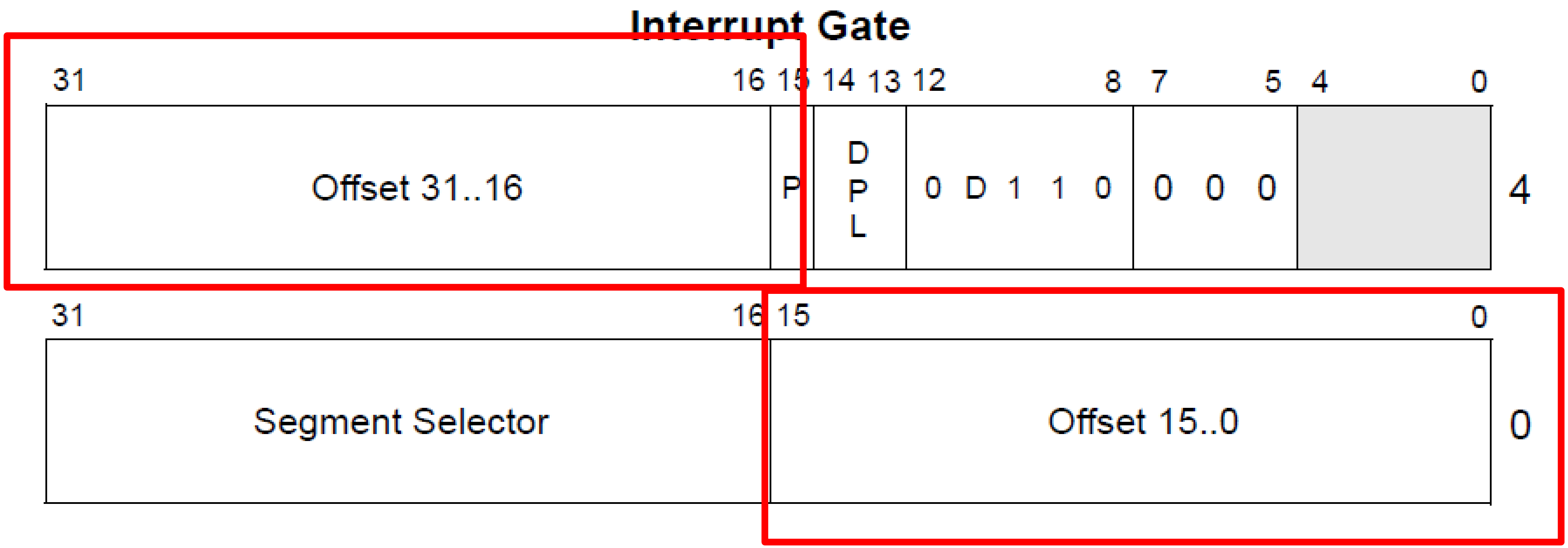



IDTR Register
CPU
47
16
15
0
IDT Base Address
IDT Limit
Interrupt
Descriptor Table (IDT)
Gate for
Interrupt #n
(n−1)∗8
Gate for
Interrupt #3
16
Gate for
Interrupt #2
8
Gate for
Interrupt #1
0
31
0
Memory

Interrupt descriptor



Interrupt descriptor

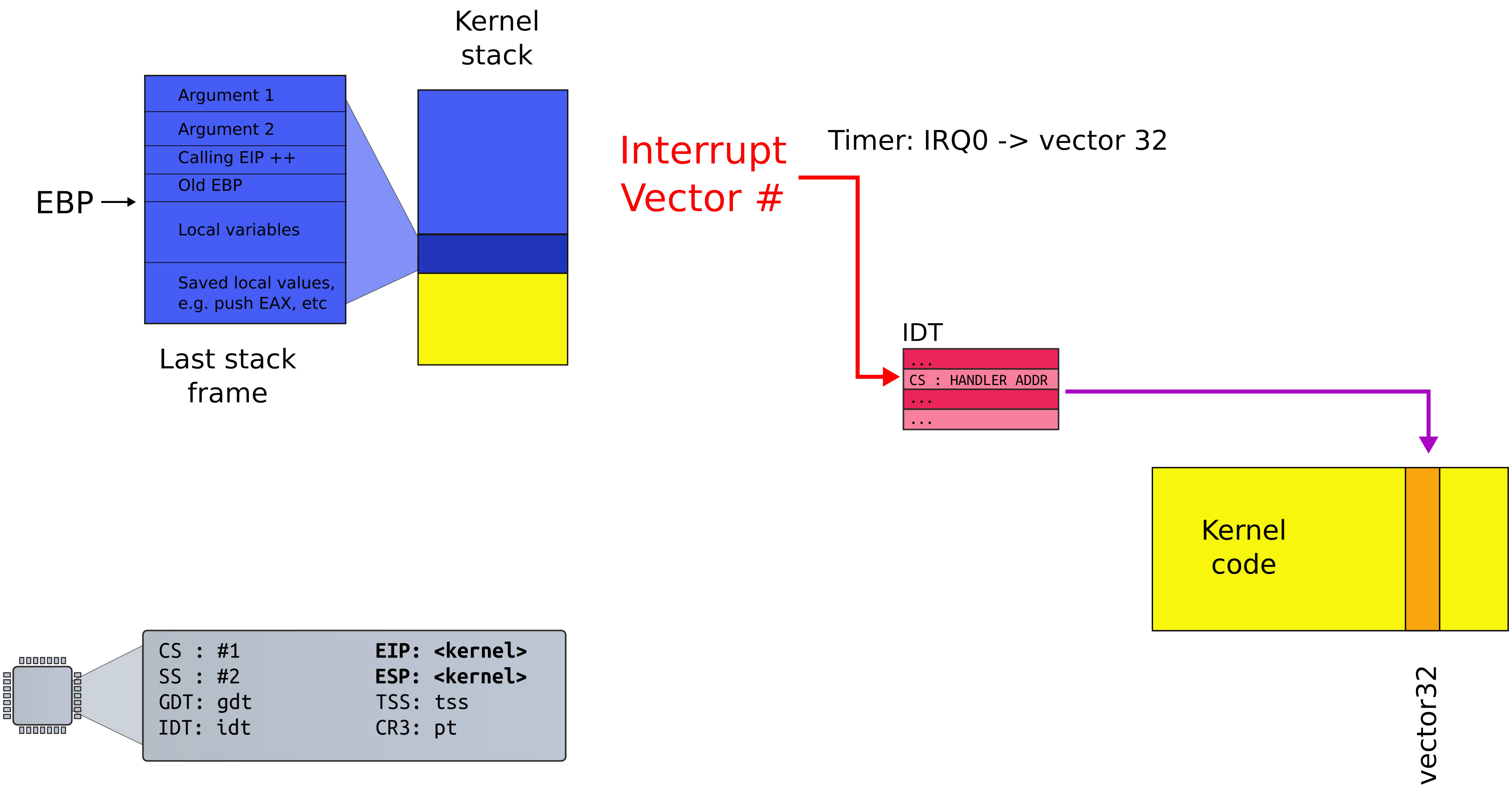


- We will walk through these fields gradually
  - For now we care about vector offset
  - Pointer to the interrupt handler

Interrupt handlers

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

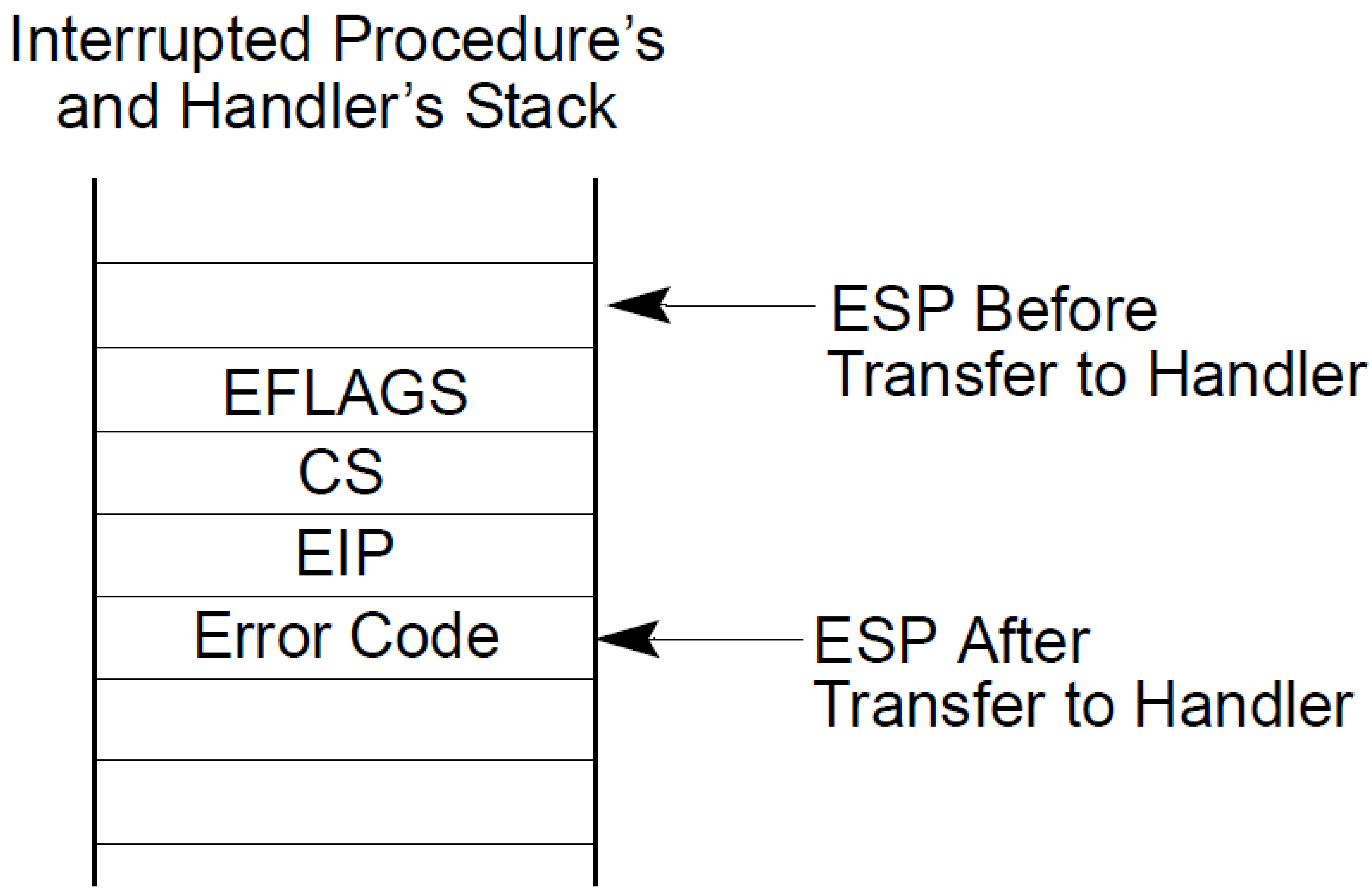
Interrupt path



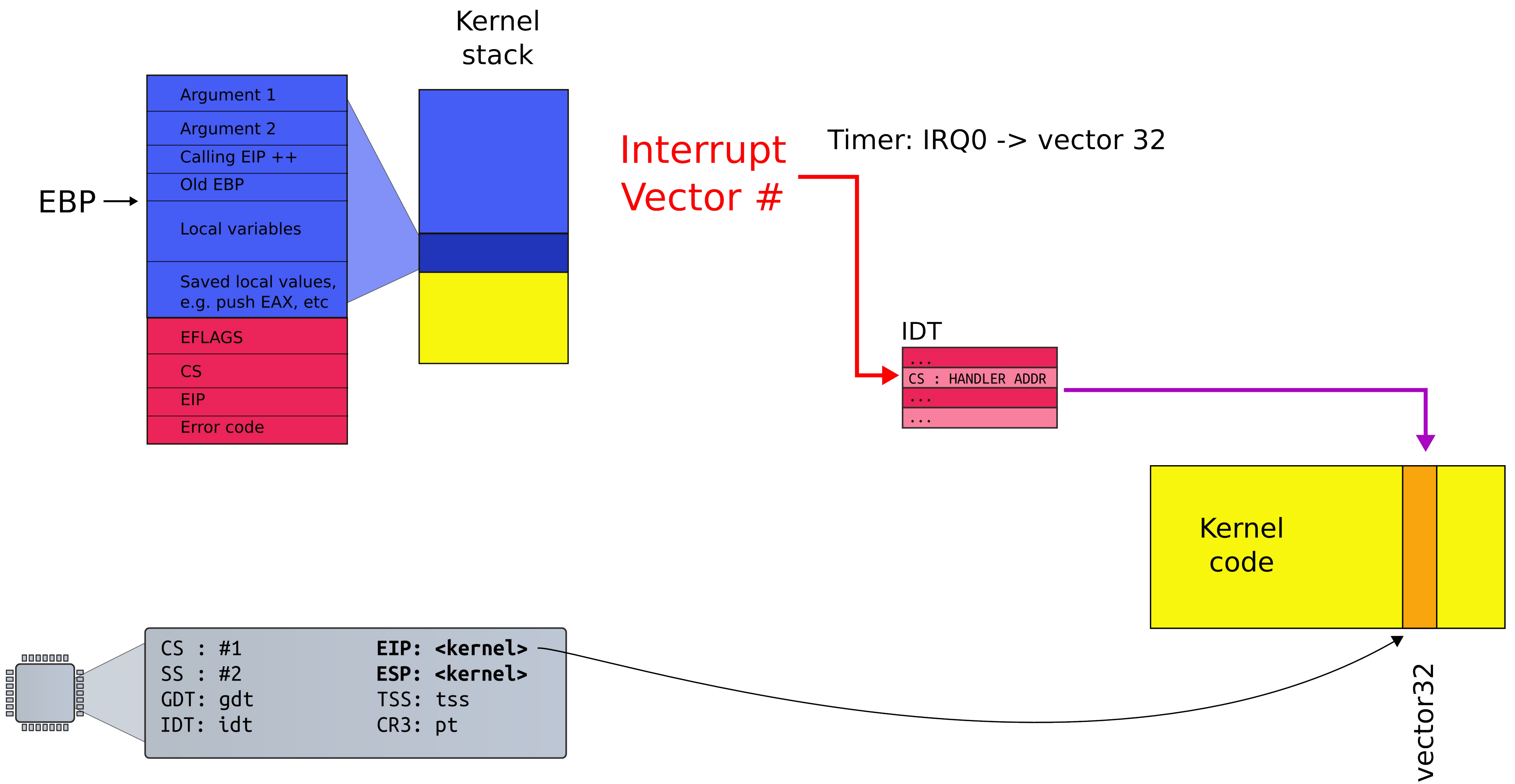
Processing of interrupt (same PL)

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



Interrupt path

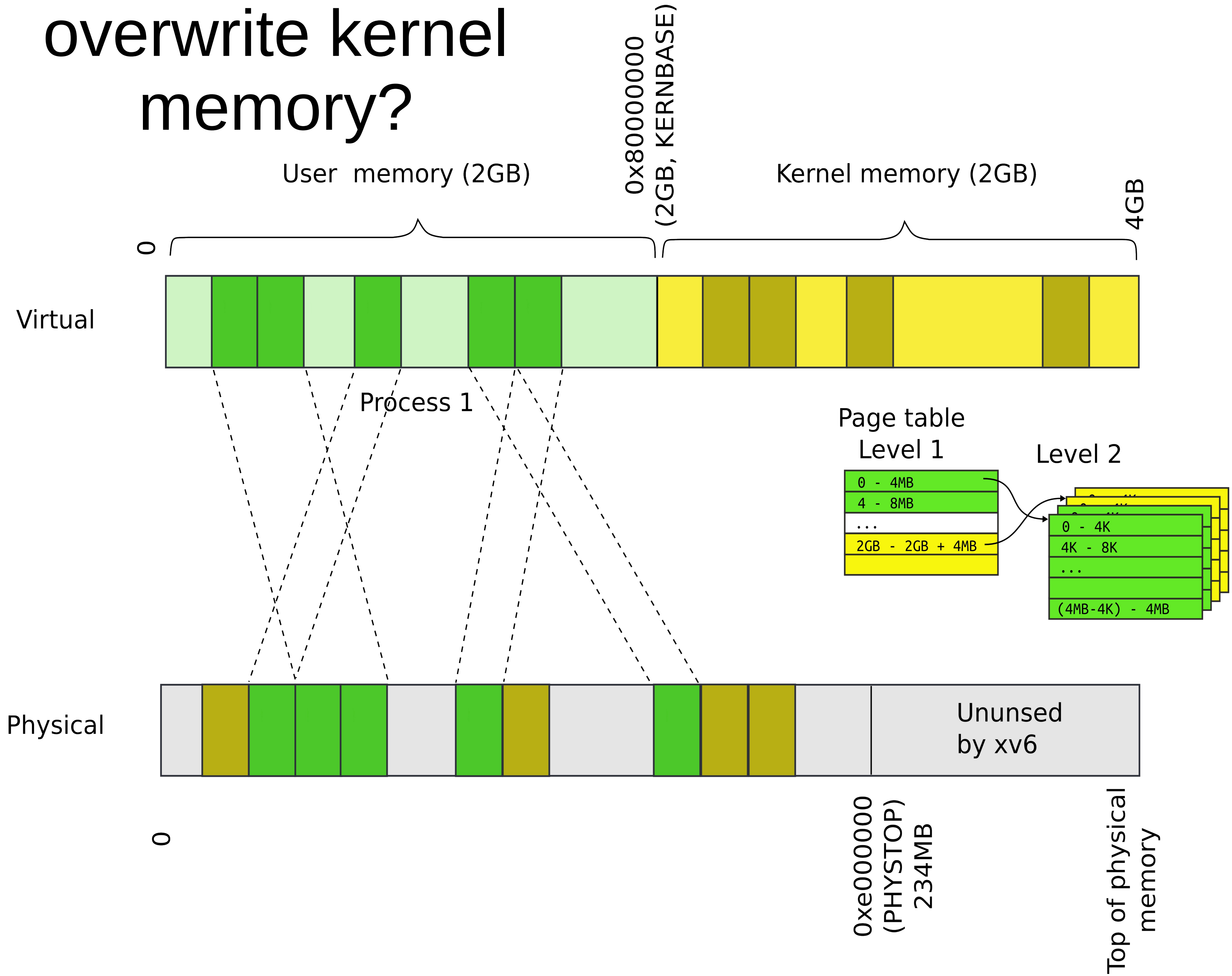


Processing of interrupt (cross PL)

- Need to change privilege level...

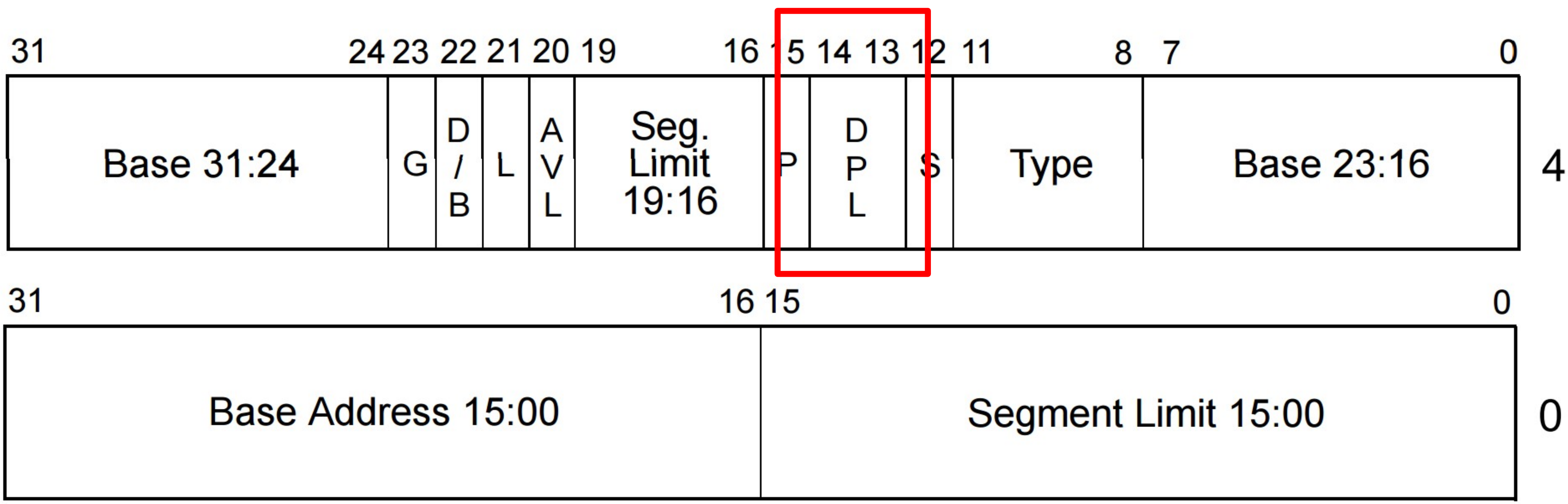
Detour:
What are those privilege levels?

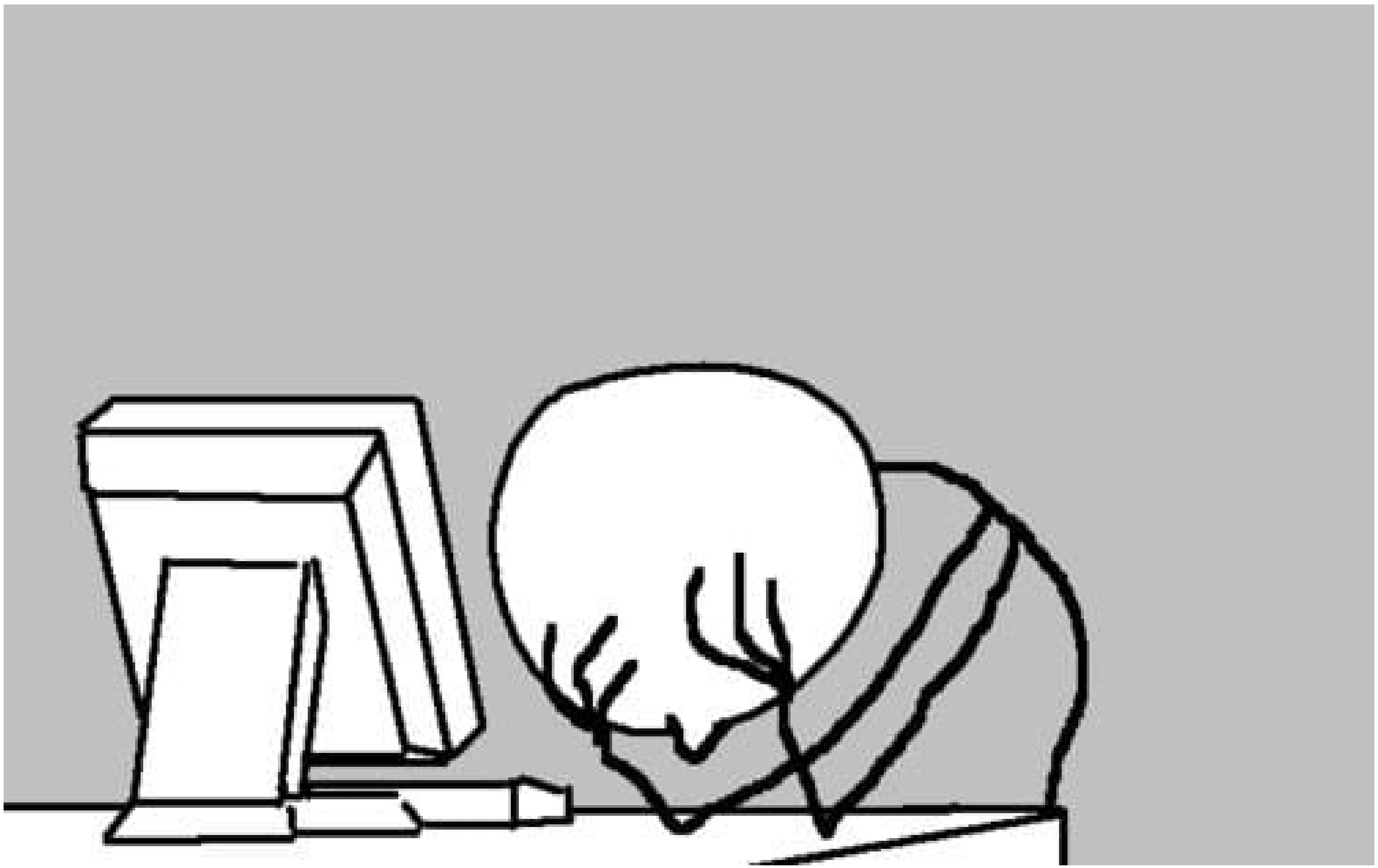
Recap: Can a process overwrite kernel memory?

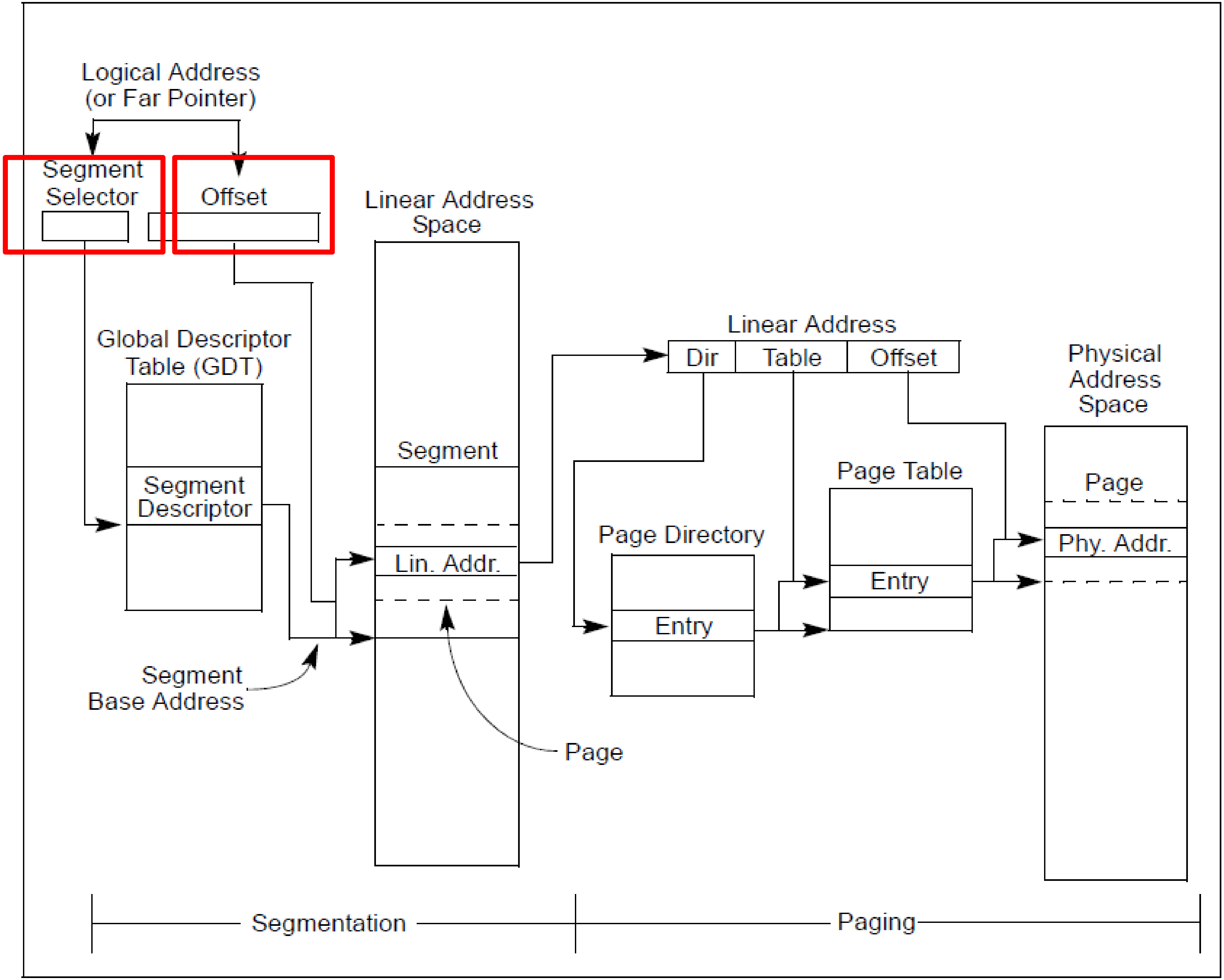


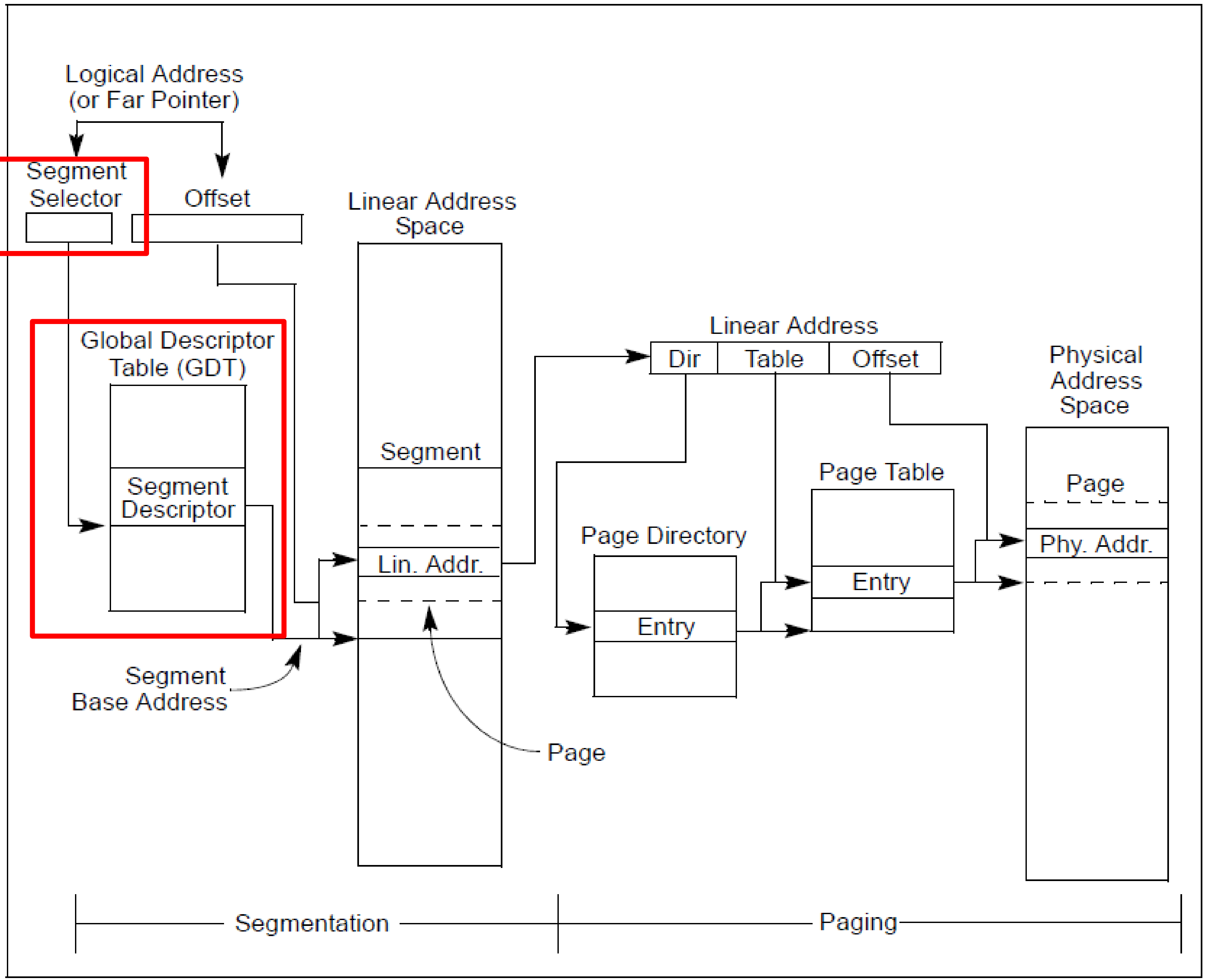
Privilege levels

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

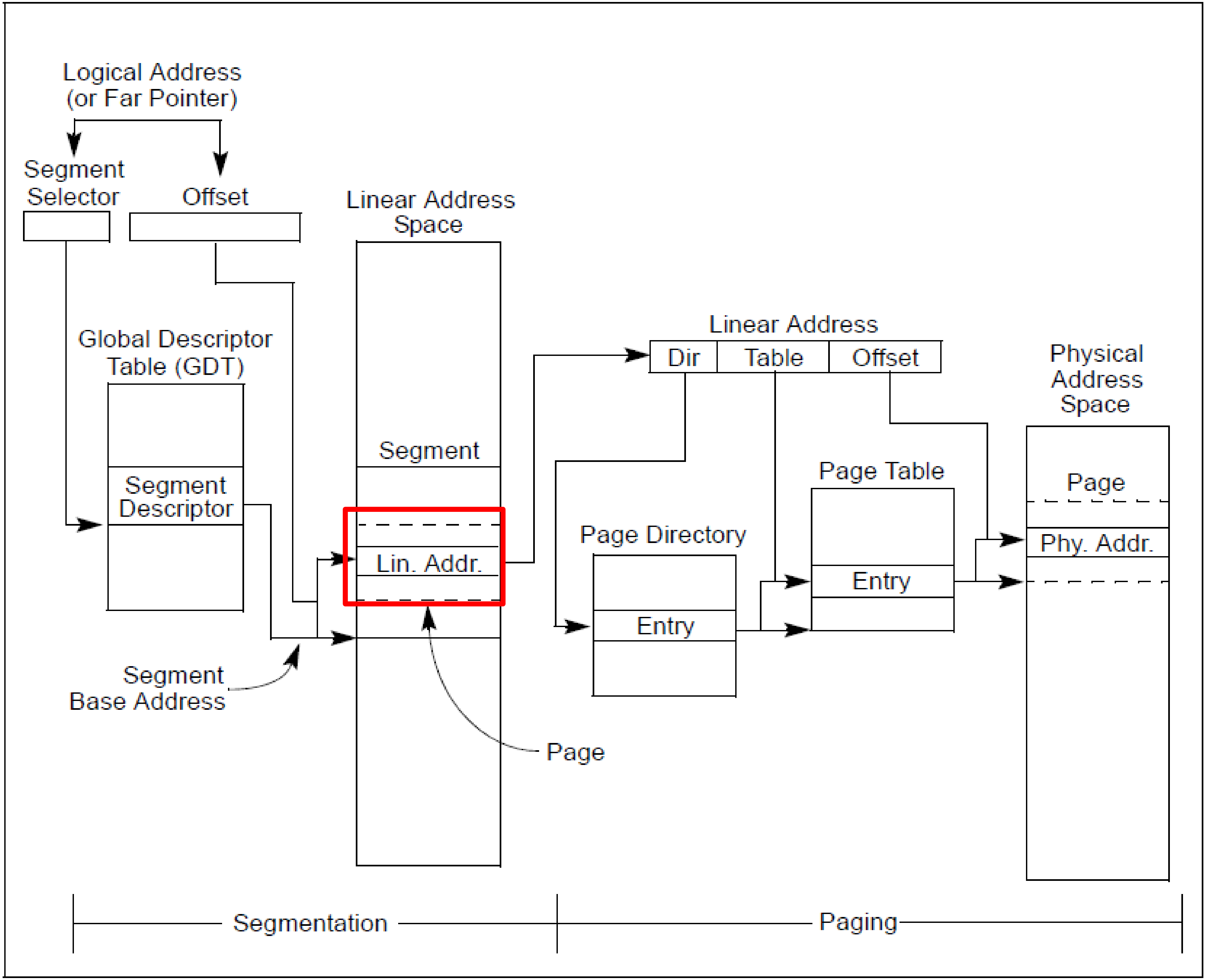




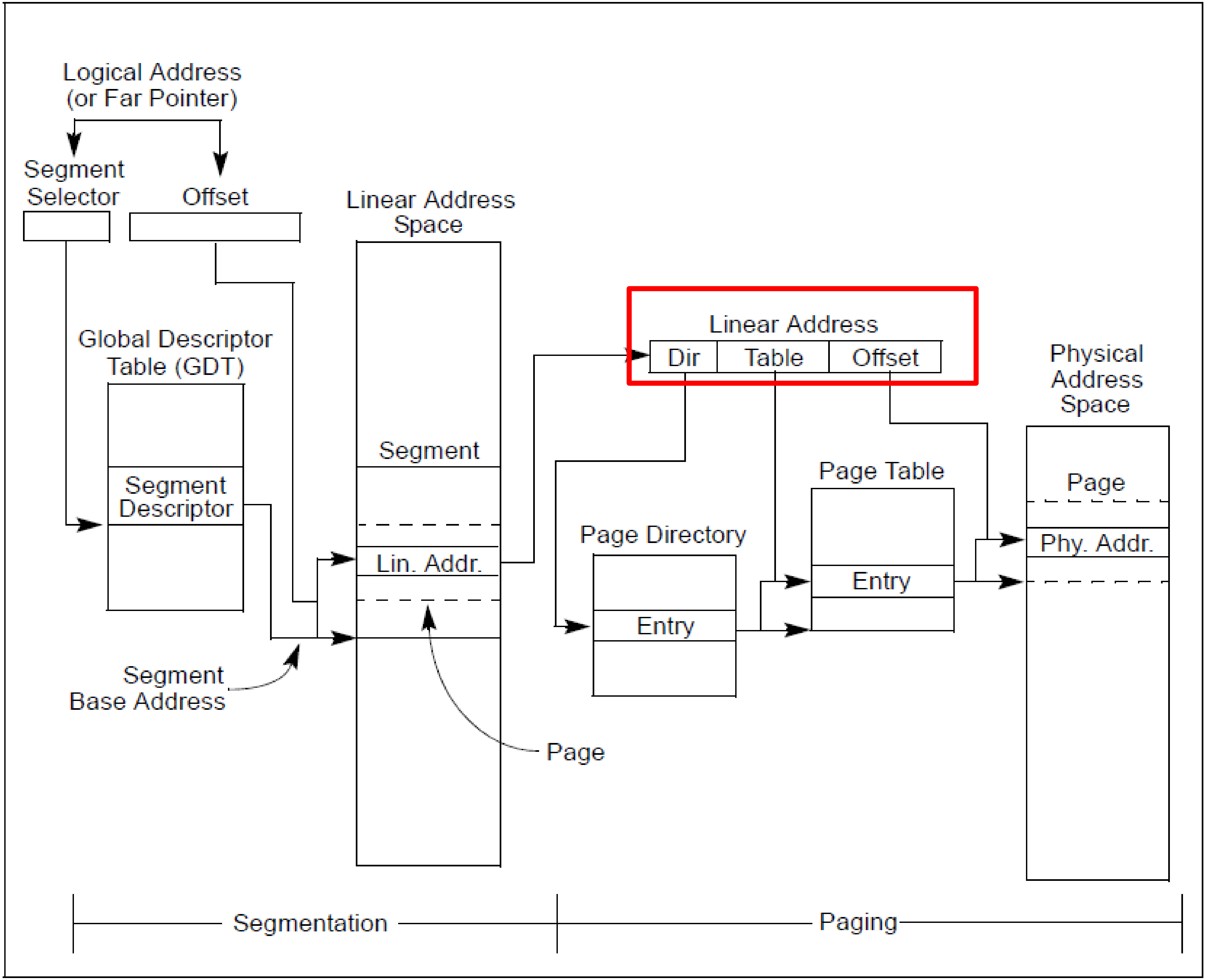

Logical Address
(or Far Pointer)
Segment
Selector
Offset
Linear Address
Space
Global Descriptor
Table (GDT)
Segment
Descriptor
Segment
Lin. Addr.
Segment
Base Address
Page
Linear Address
Dir
Table
Offset
Page Directory
Entry
Page Table
Entry
Physical
Address
Space
Page
Phy. Addr.
Segmentation
Paging



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



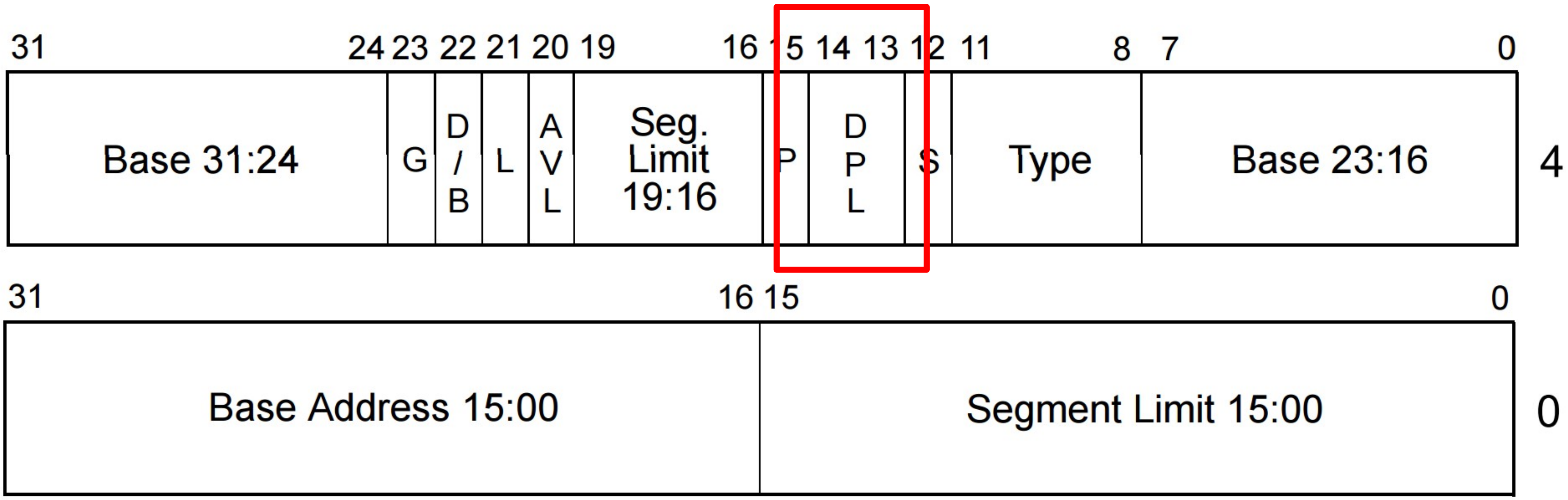
Logical Address
(or Far Pointer)
Segment
Selector
Offset
Linear Address
Space
Global Descriptor
Table (GDT)
Segment
Descriptor
Segment
Lin. Addr.
Linear Address
Dir
Table
Offset
Physical
Address
Space
Page Table
Page
Page Directory
Phy. Addr.
Entry
Entry
Segment
Base Address
Page
Segmentation
Paging



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



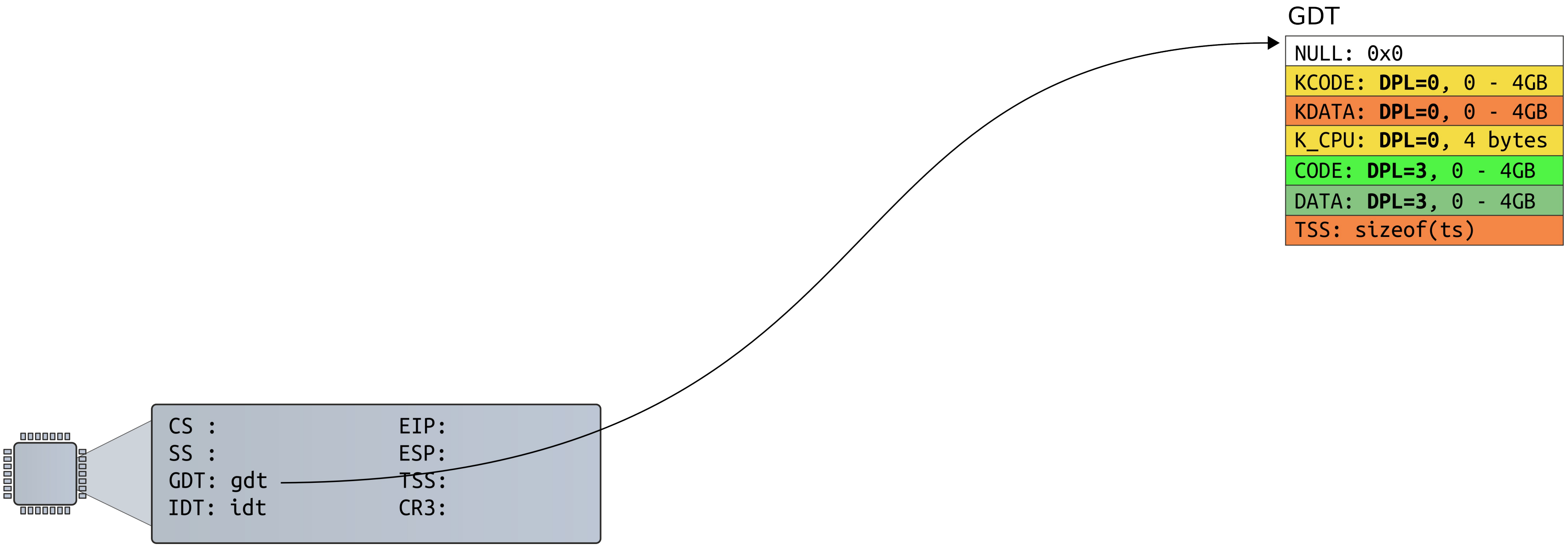
Privilege levels

- 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

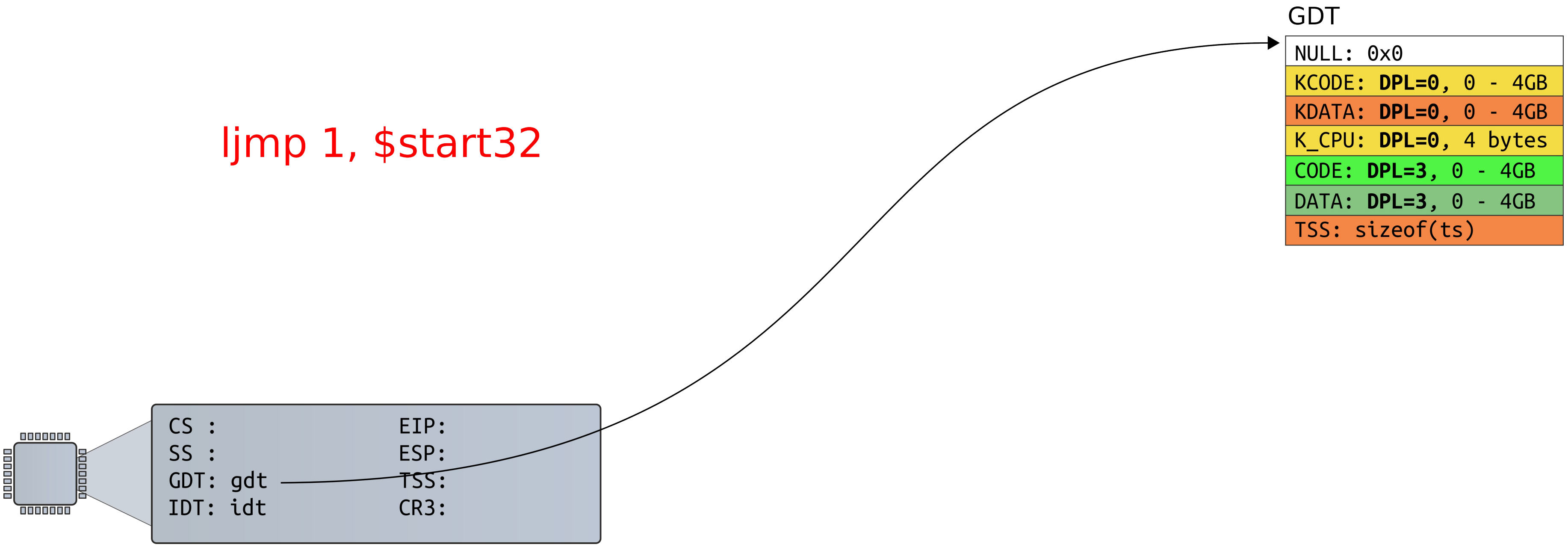
Privilege level transitions

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

Xv6 example: started boot (no CPL yet)

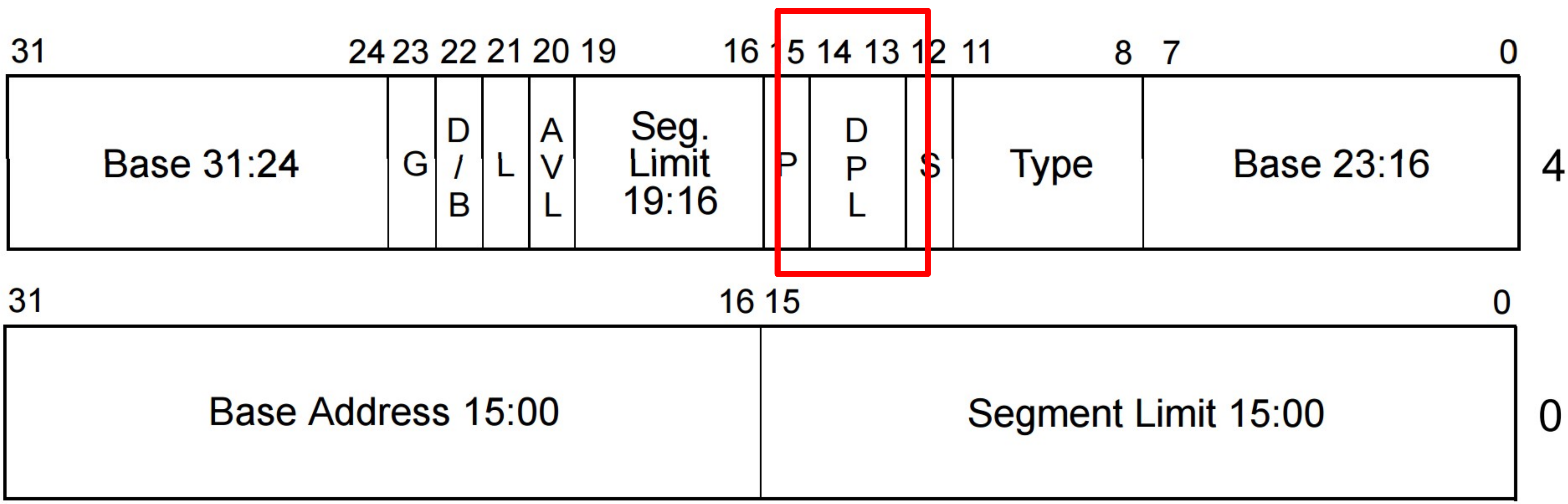


Xv6 example: prepare to load GDT entry #1

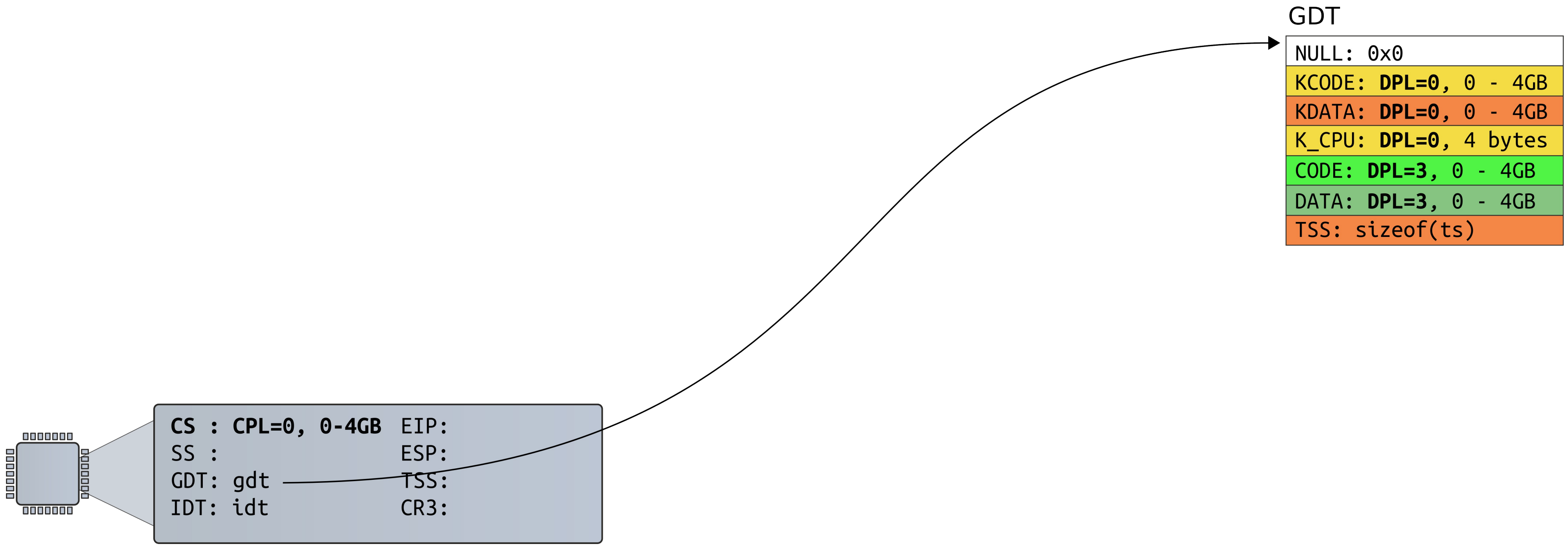


Privilege levels

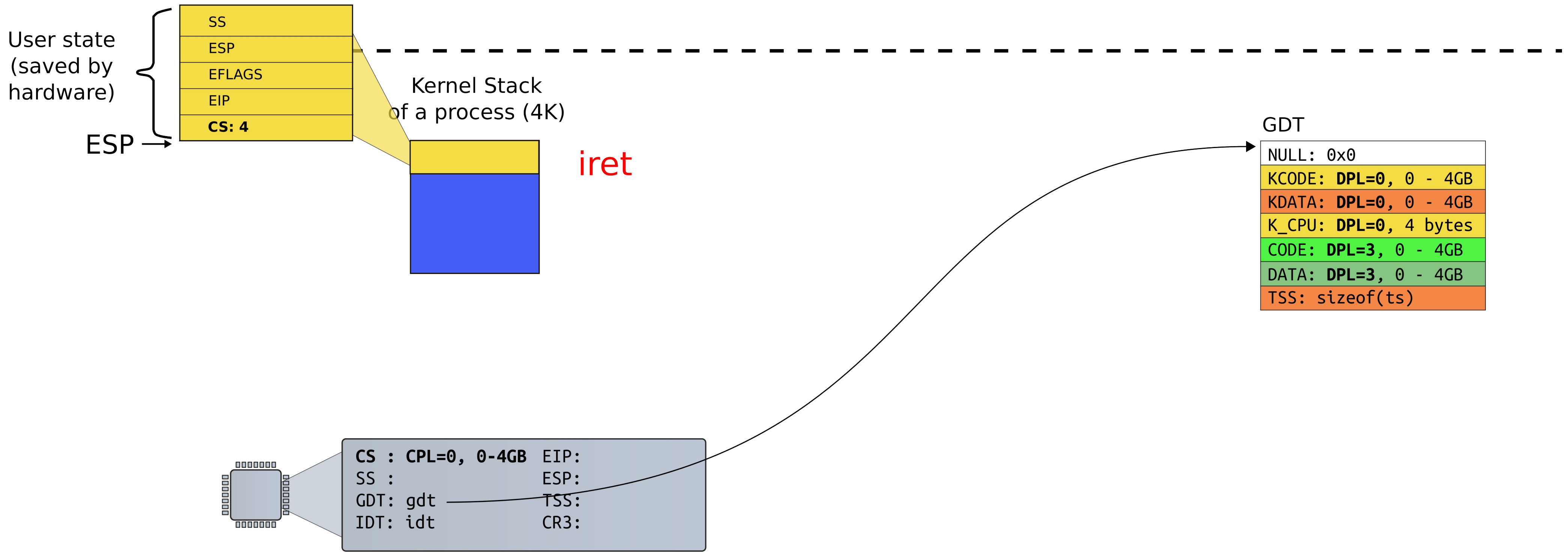
- Each segment has a privilege level
  - DPL (descriptor privilege level)
  - 4 privilege levels ranging 0-3



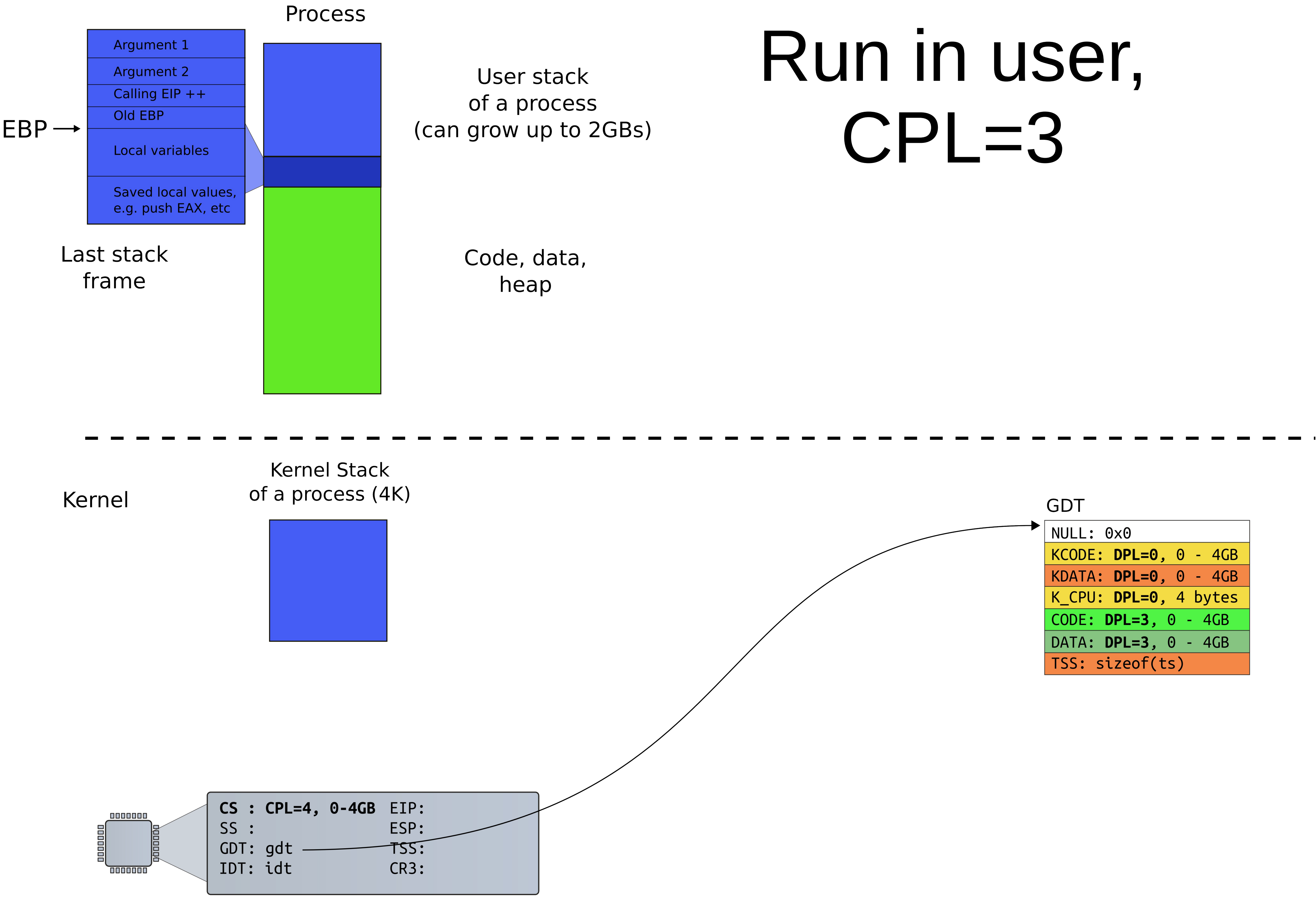
Now CPL=0. We run in the kernel



iret: return to user, load GDT #4



Run in user, CPL=3

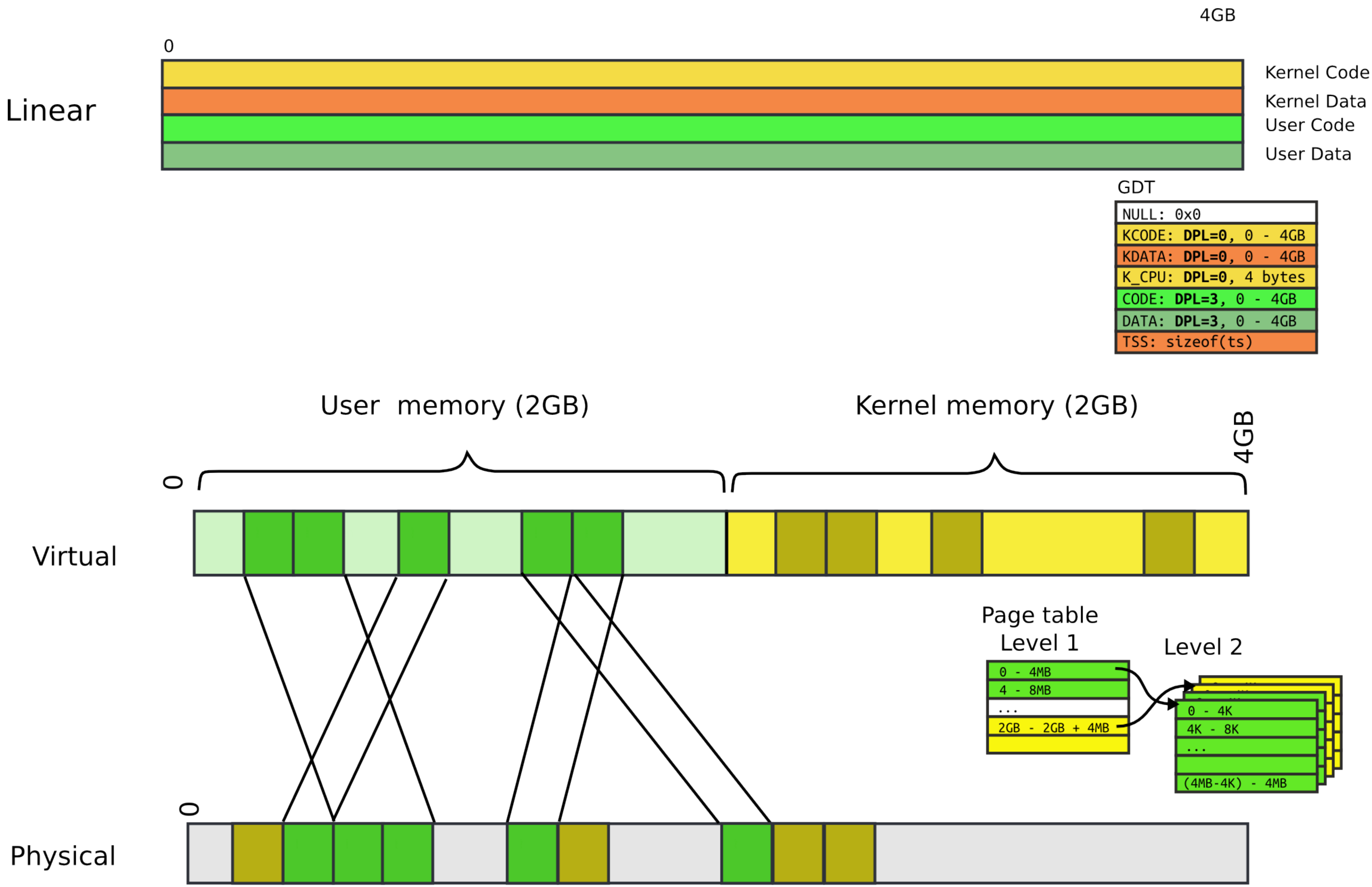


Real world

- 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?

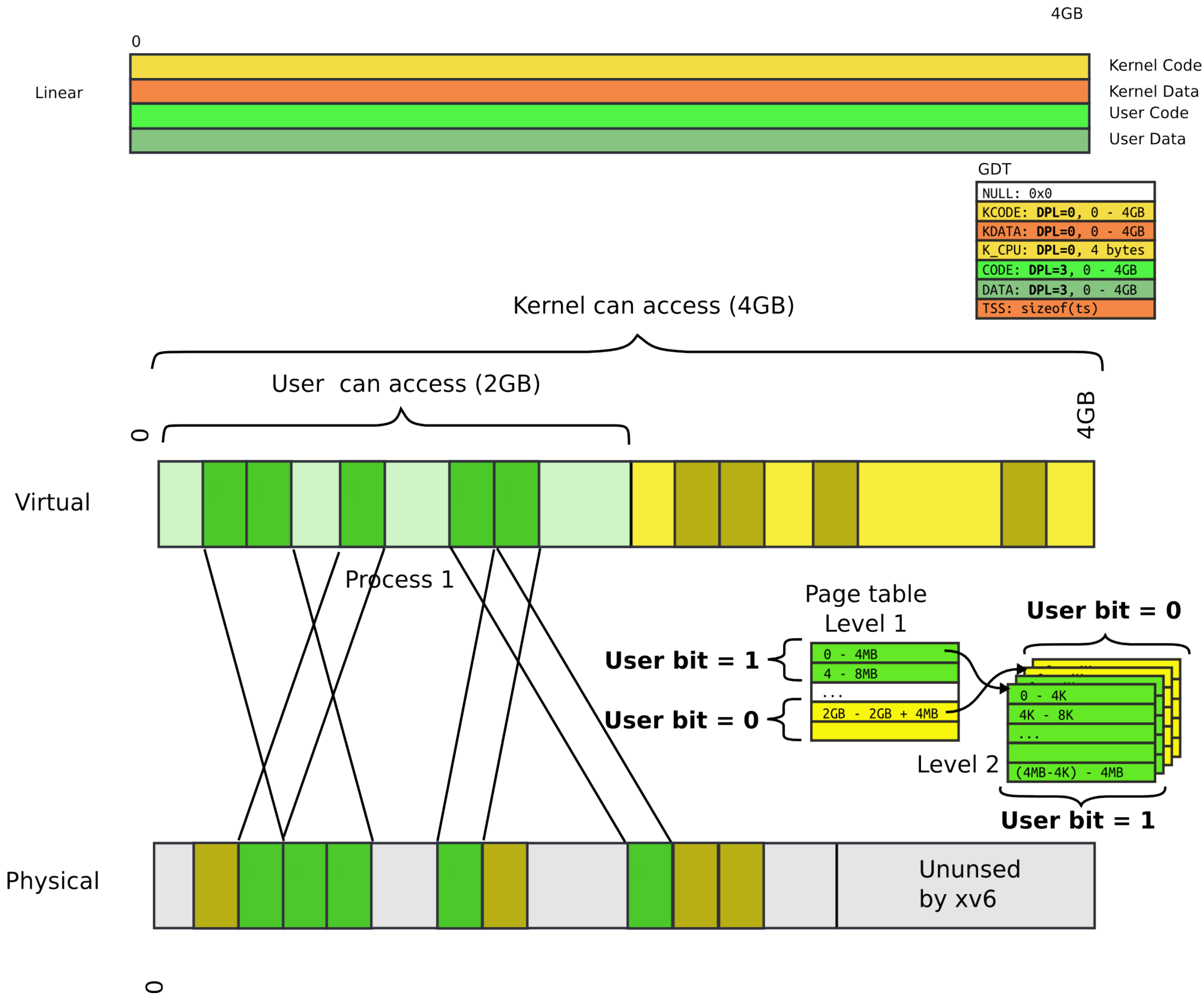
Real world

- 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?
  - **Page tables**


Linear
0
4GB
Kernel Code
Kernel Data
User Code
User Data
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)
User memory (2GB)
Kernel memory (2GB)
0
4GB
Virtual
Page table
Level 1
0 - 4MB
4 - 8MB
...
2GB - 2GB + 4MB
Level 2
0 - 4K
4K - 8K
...
(4MB-4K) - 4MB
0
Physical

Page table: user bit

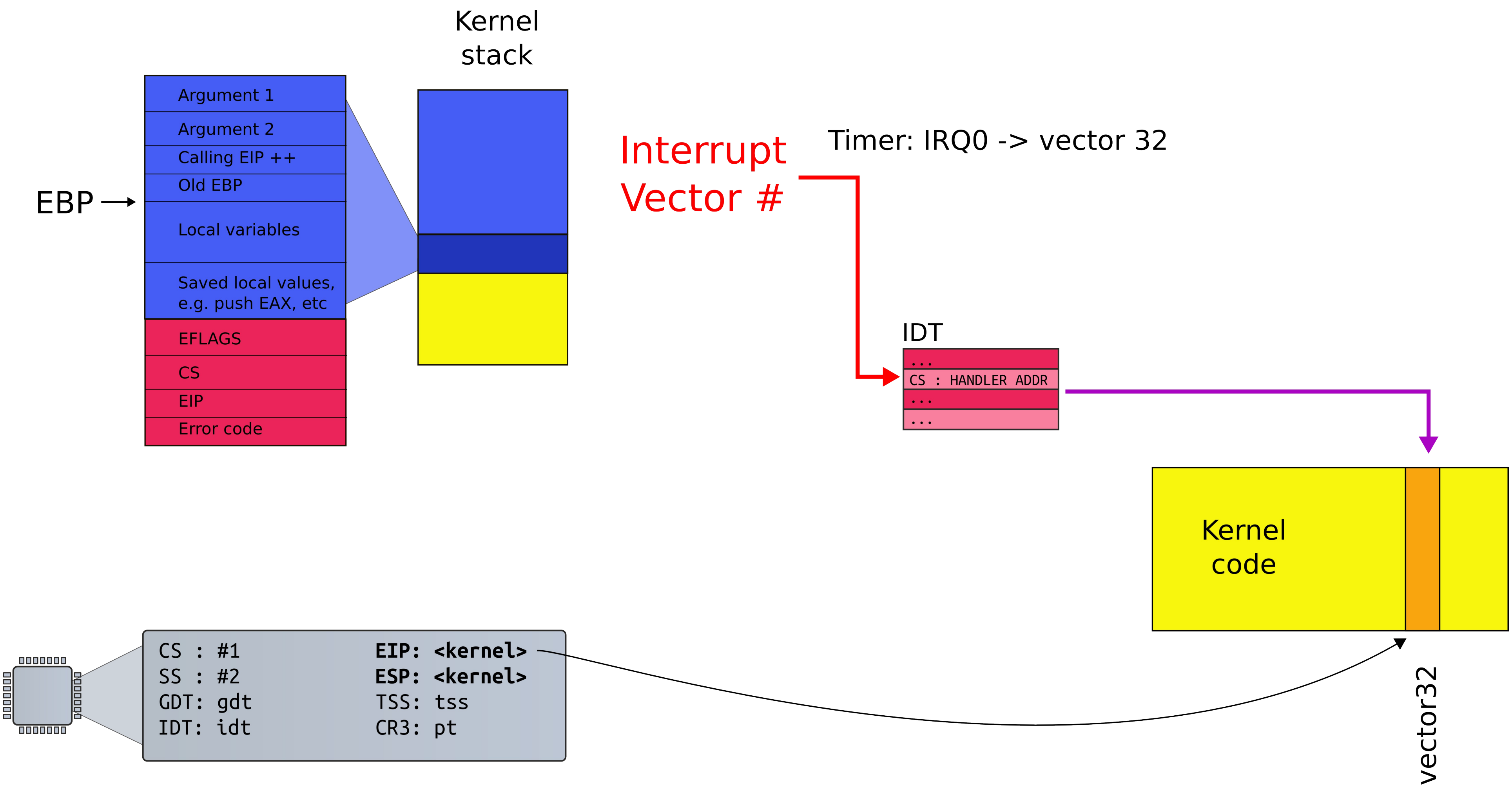
- Each entry (both Level 1 and Level 2) has a bit
  - If set, code at privilege level 3 can access
  - If not, only levels 0-2 can access
- Note, only 2 levels, not 4 like with segments
- All kernel code is mapped with the user bit clear
  - This protects user-level code from accessing the kernel



4GB
0
Linear
Kernel Code
Kernel Data
User Code
User Data
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)
Kernel can access (4GB)
User can access (2GB)
0
4GB
Virtual
Process 1
Page table
Level 1
User bit = 1
User bit = 0
0 - 4MB
4 - 8MB
...
2GB - 2GB + 4MB
User bit = 0
0 - 4K
4K - 8K
...
(4MB-4K) - 4MB
Level 2
User bit = 1
Physical
Ununsed
by xv6
0

End of detour:
Back to handling interrupts

Recap: interrupt path, no PL change

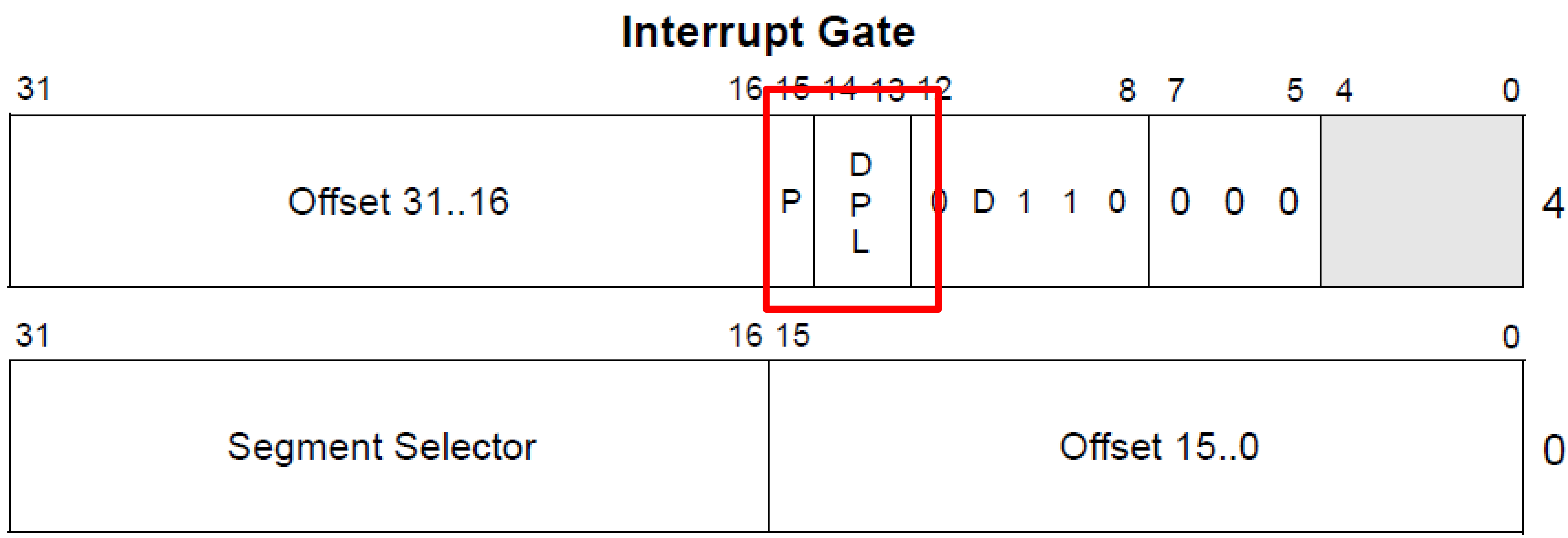


Processing of an interrupt when change of a privilege level is required

Processing of interrupt (cross PL)

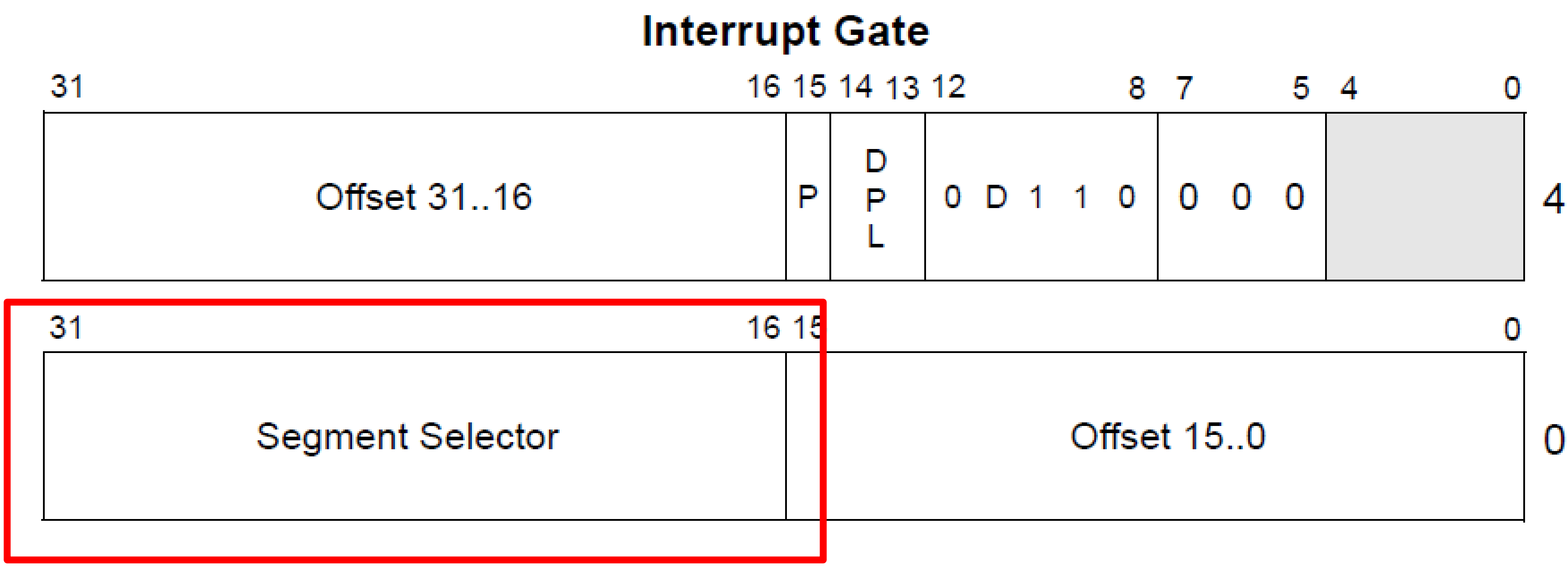
- Assume we're at CPL =3 (user)

Interrupt descriptor (an entry in the IDT)



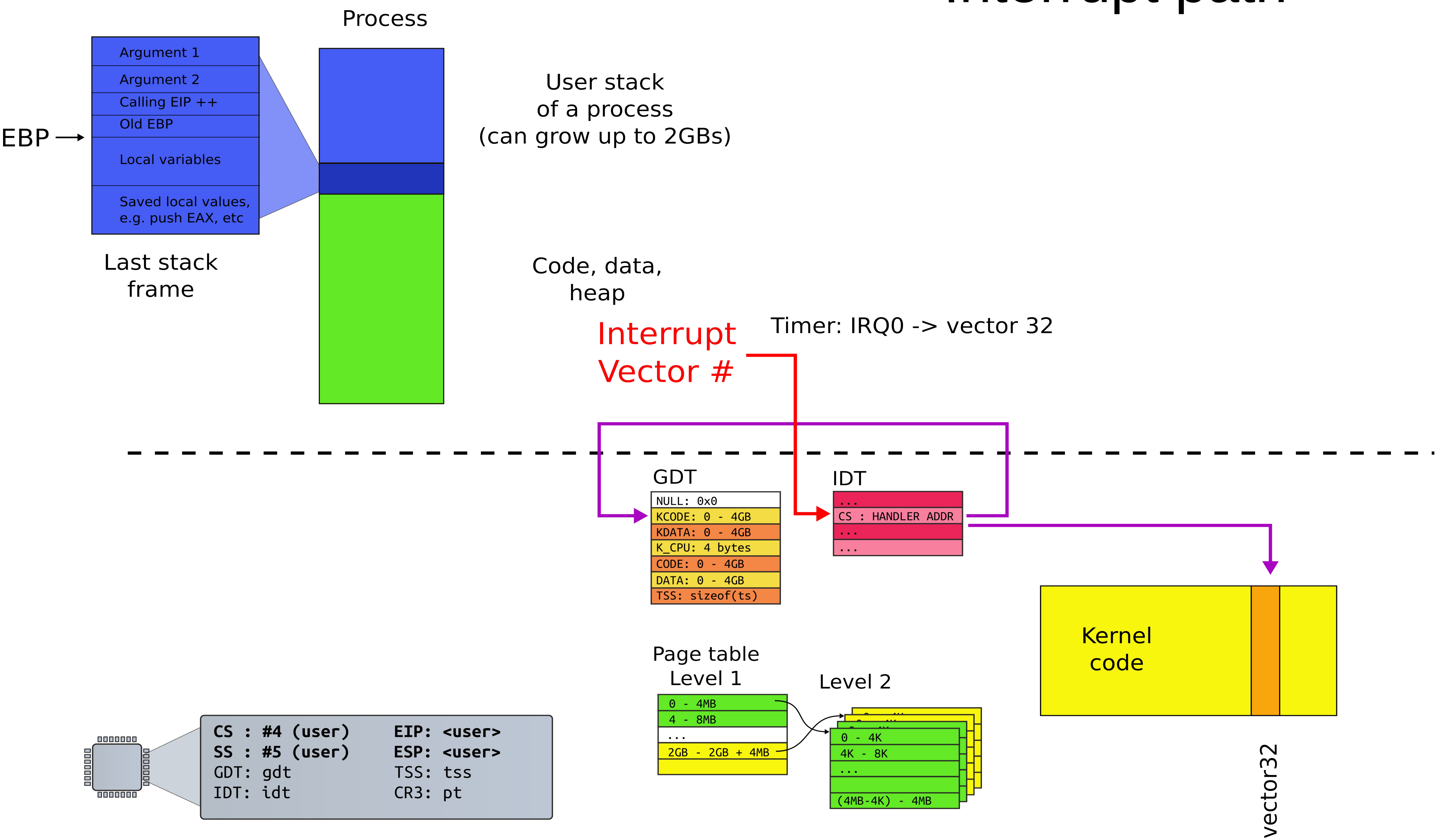
- Interrupt is allowed if...
  - current privilege level (CPL) is less or equal to descriptor privilege level (DPL)
- The kernel protects device interrupts from user

Interrupt descriptor (an entry in the IDT)



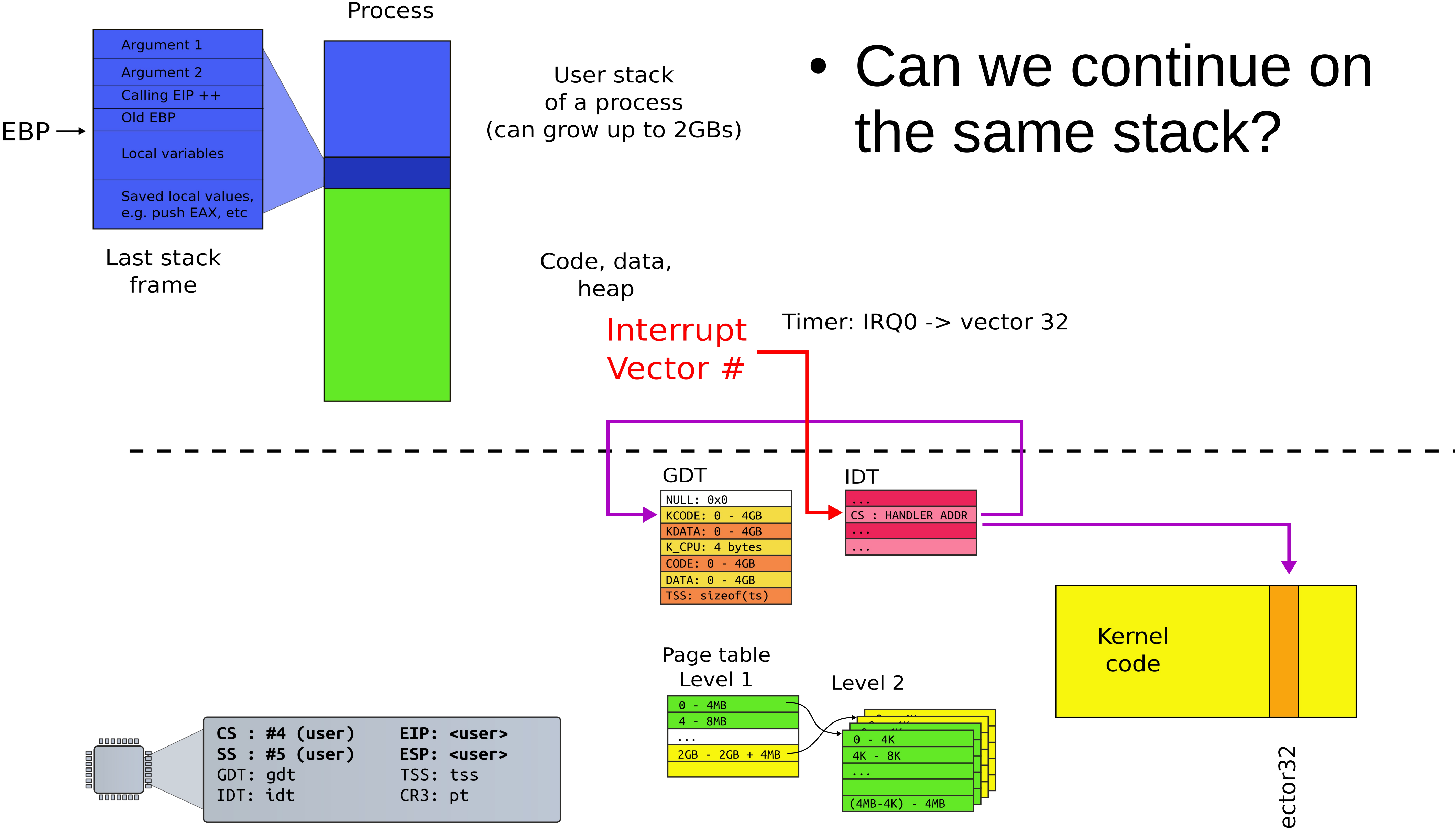
- Note that this new segment can be more privileged
  - E.g., CPL = 3, DPL = 3, new segment can be PL = 0
  - This is how user-code (PL=3) transitions into kernel (PL=0)

Interrupt path



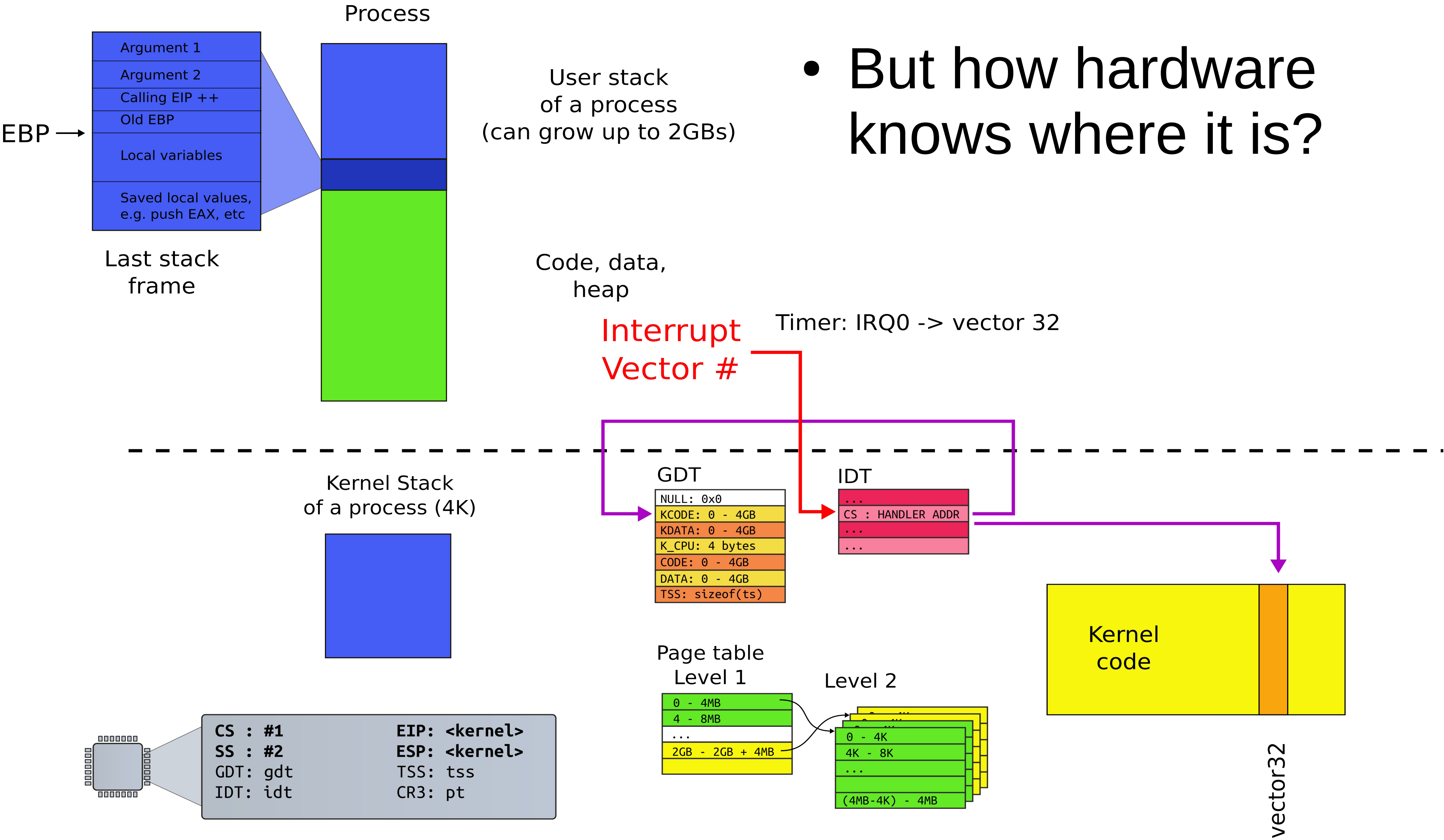
Stack

- Can we continue on the same stack?

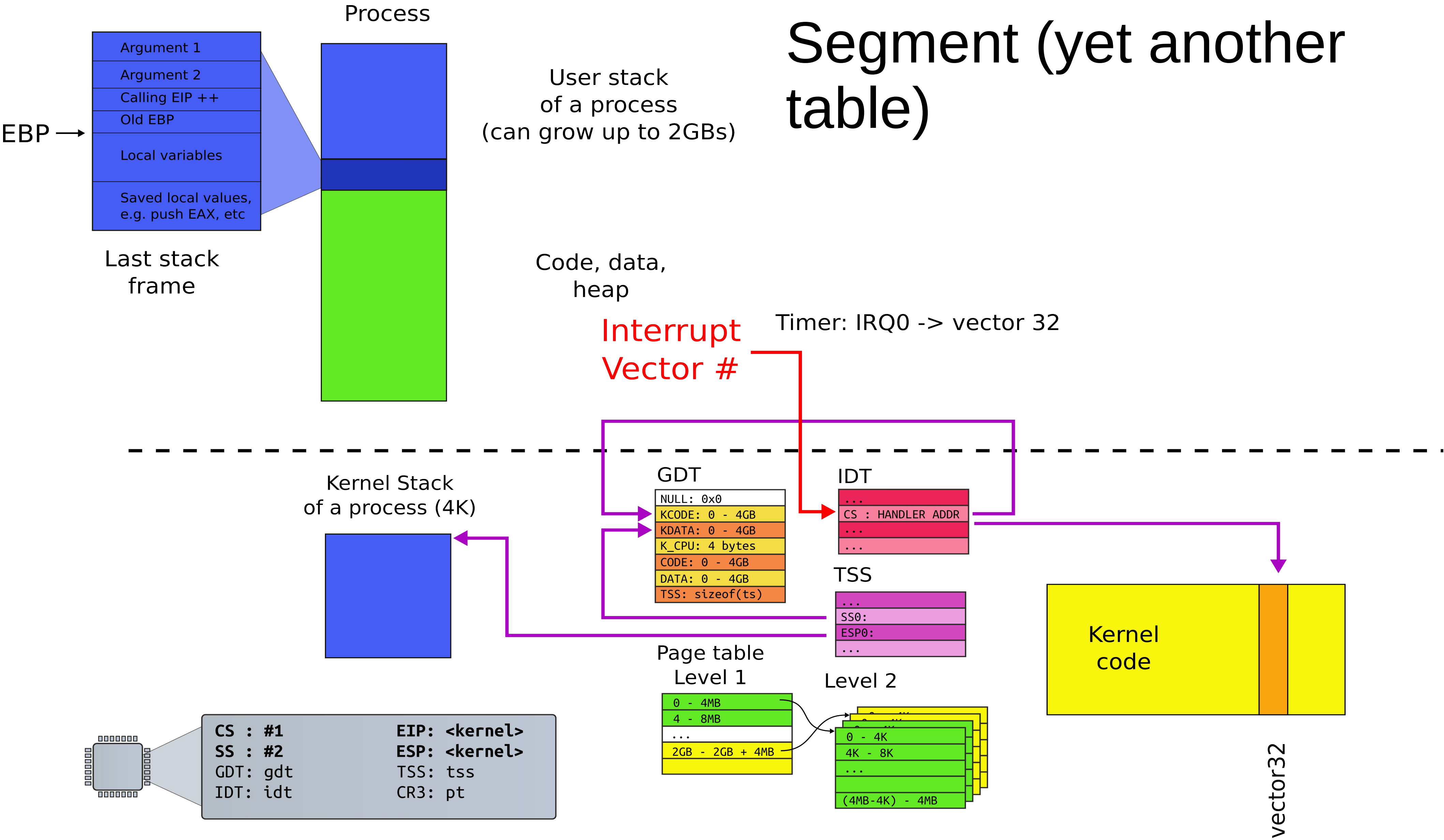


Stack

- But how hardware knows where it is?



TSS: Task State Segment (yet another table)



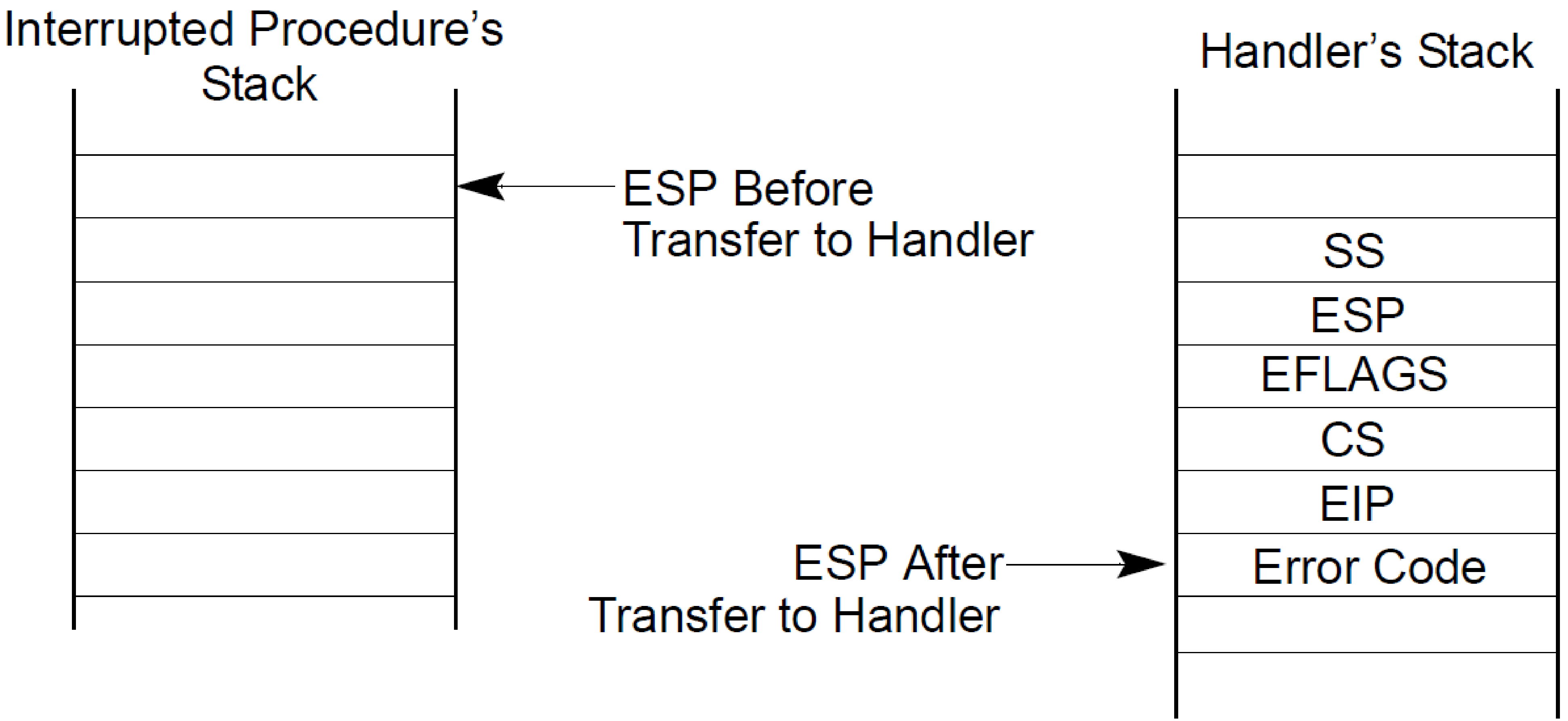
Task State Segment

- 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

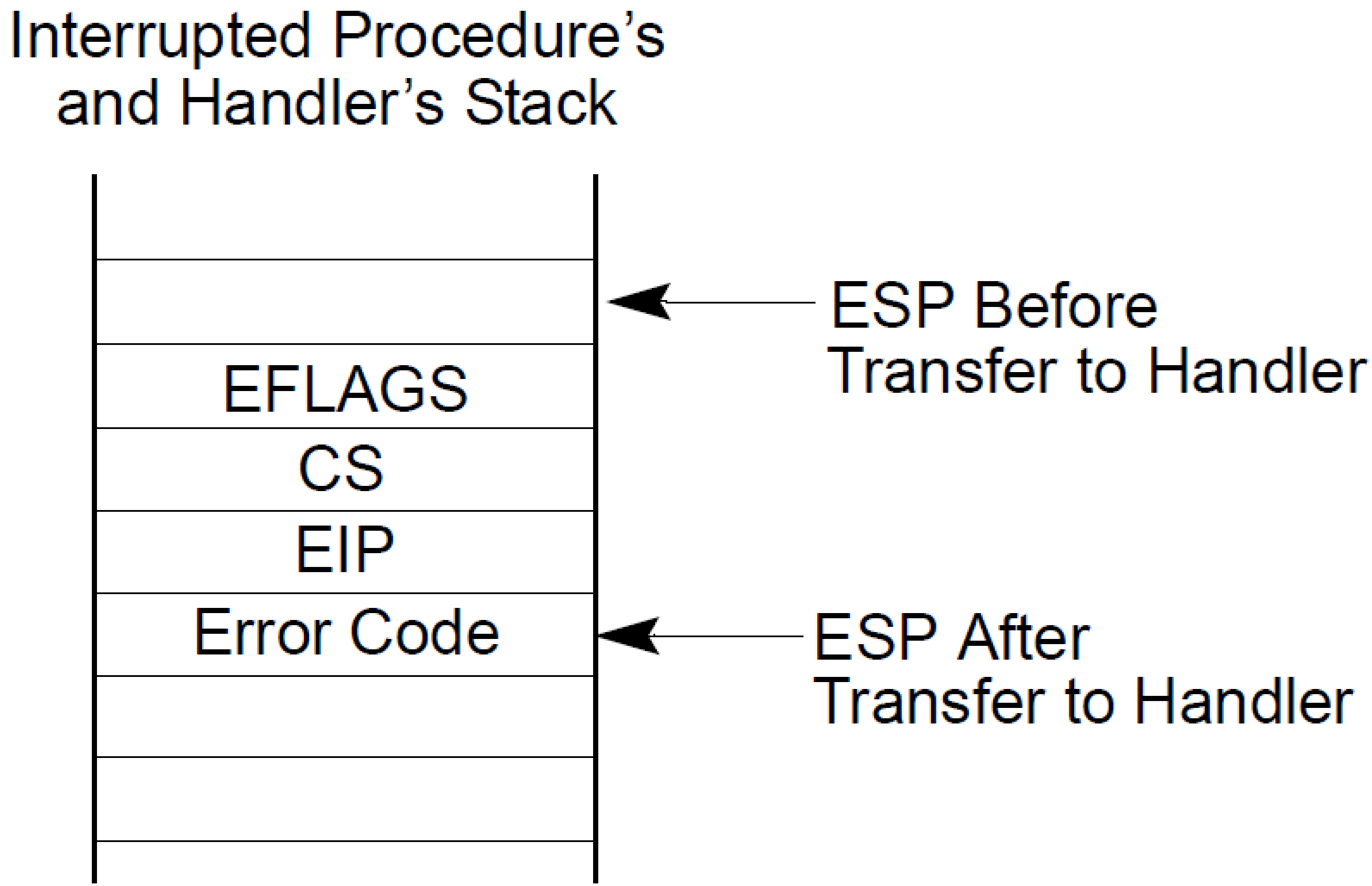
Processing of interrupt (cross PL)

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

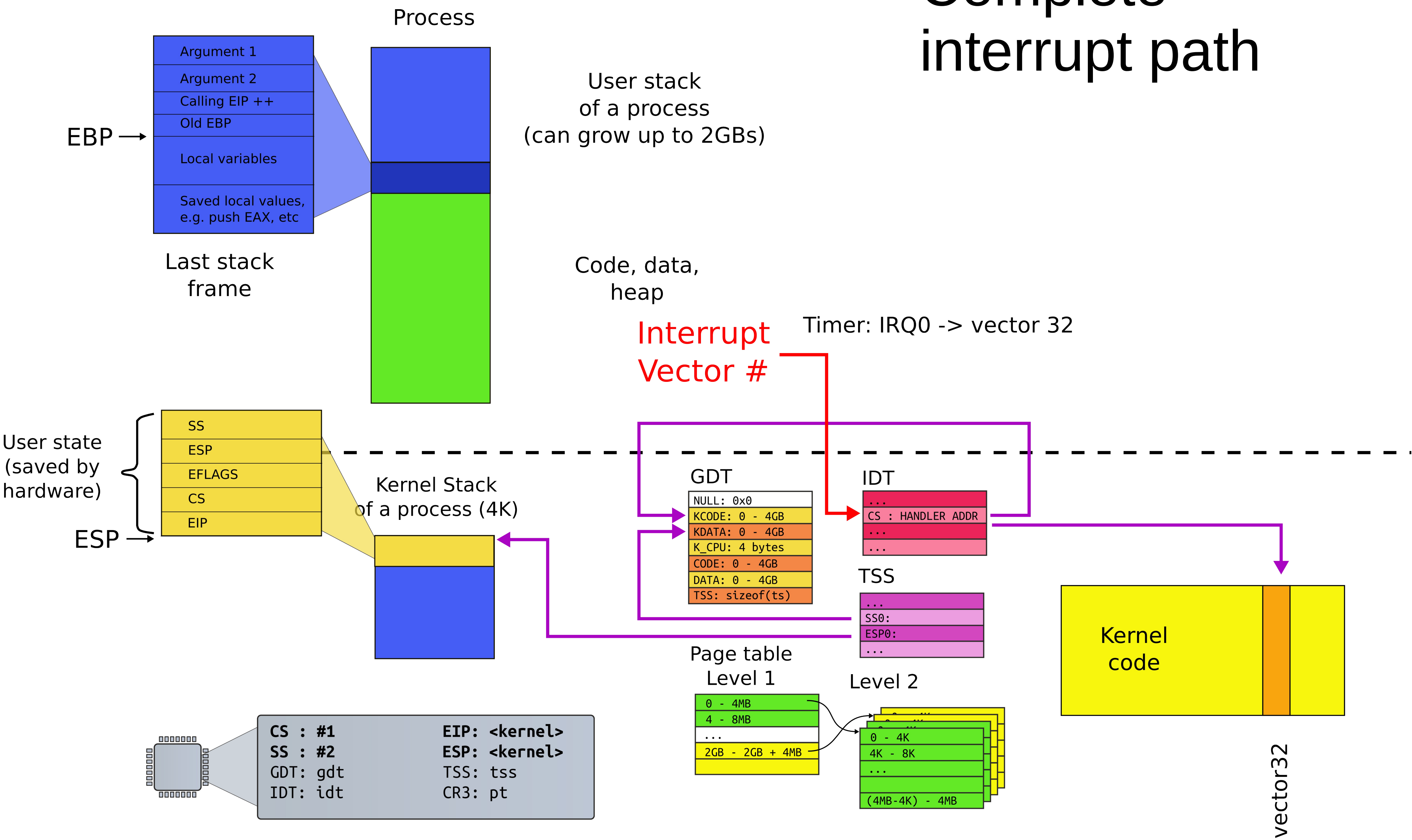
Stack Usage with Privilege-Level Change



Stack Usage with No Privilege-Level Change



Complete interrupt path



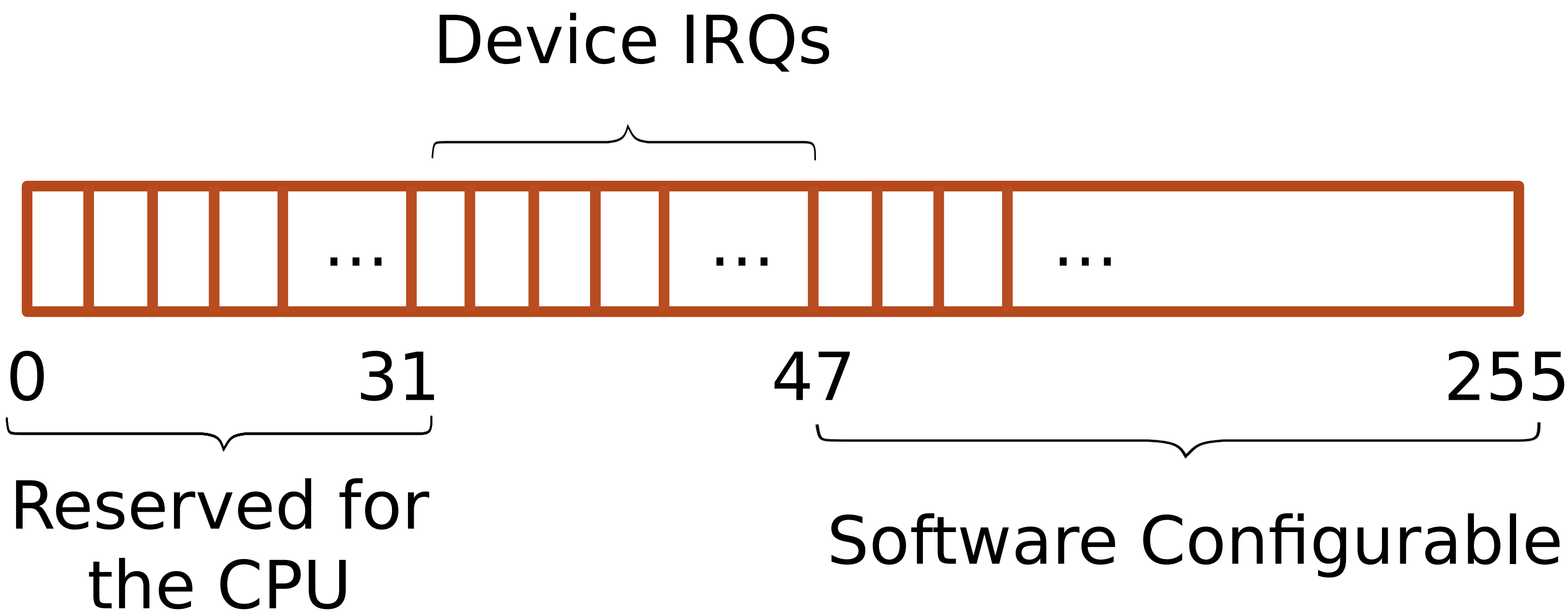
Return from an interrupt

- 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

Interrupt descriptor table (IDT)

x86 interrupt descriptor table



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)
  - Most device IRQ lines can be configured
  - Look up APICs for more info (Ch 4 of Bovet and Cesati)
  - 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)

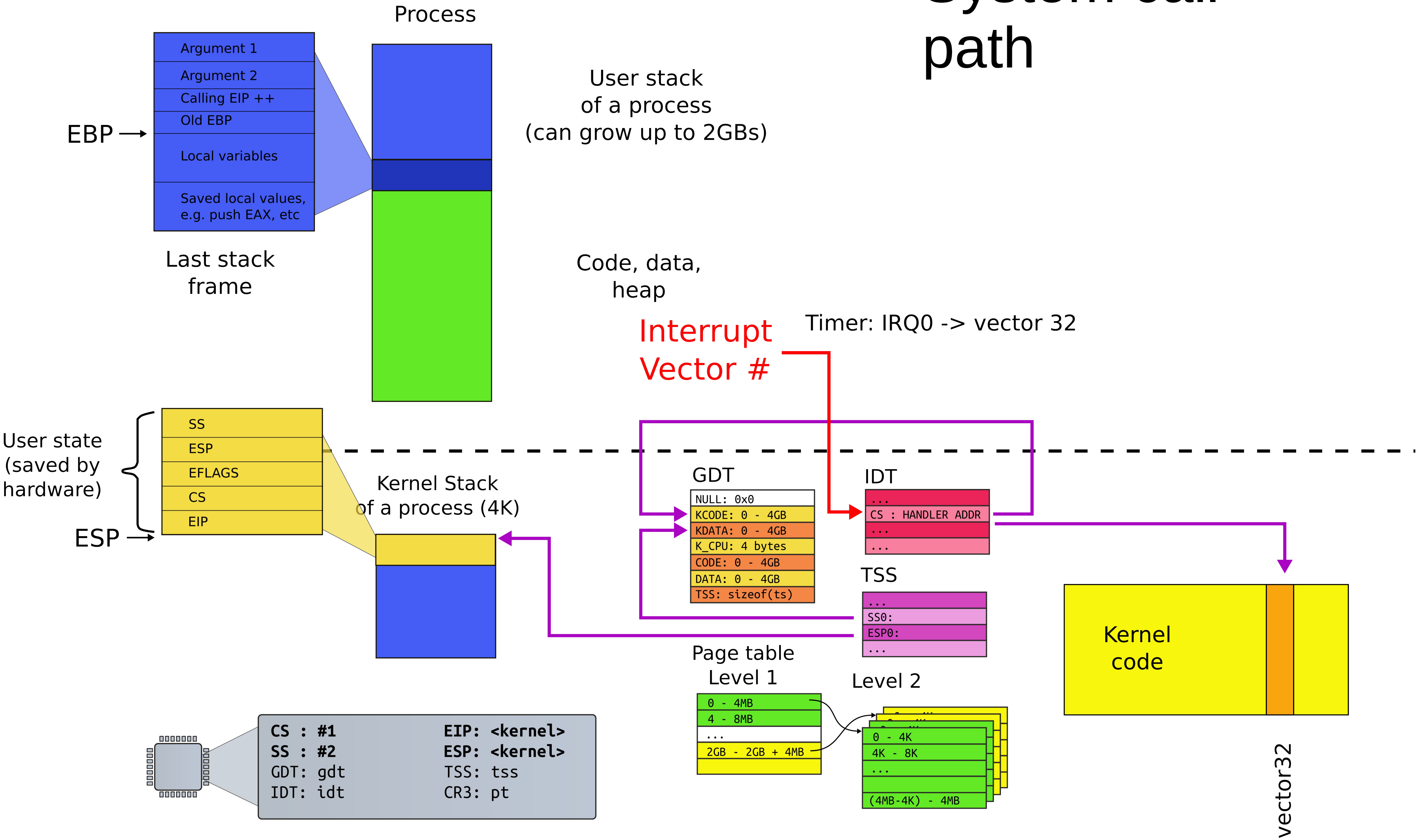
- Delivered even if IF is clear, e.g. interrupts disabled
  - CPU blocks subsequent NMI interrupts until IRET
- Sources
  - External hardware asserts the NMI pin
  - Processor receives a message on the system bus, or the APIC serial bus with NMI delivery mode
- Delivered via vector #2

System calls

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

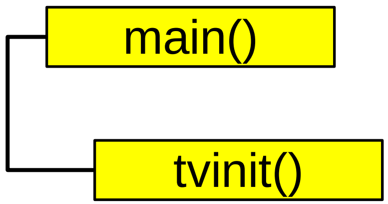
System call path



Initialize IDT

- tvinit() is called from main()

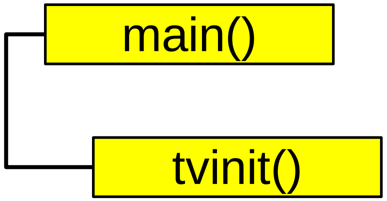
```
void
tvinit(void)
{
  int i;

  for(i = 0; i < 256; i++)
    SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
  SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3,
              vectors[T_SYSCALL], DPL_USER);

  initlock(&tickslock, "time");
}
```



Initialize IDT

- System call interrupt vector (T_SYSCALL)

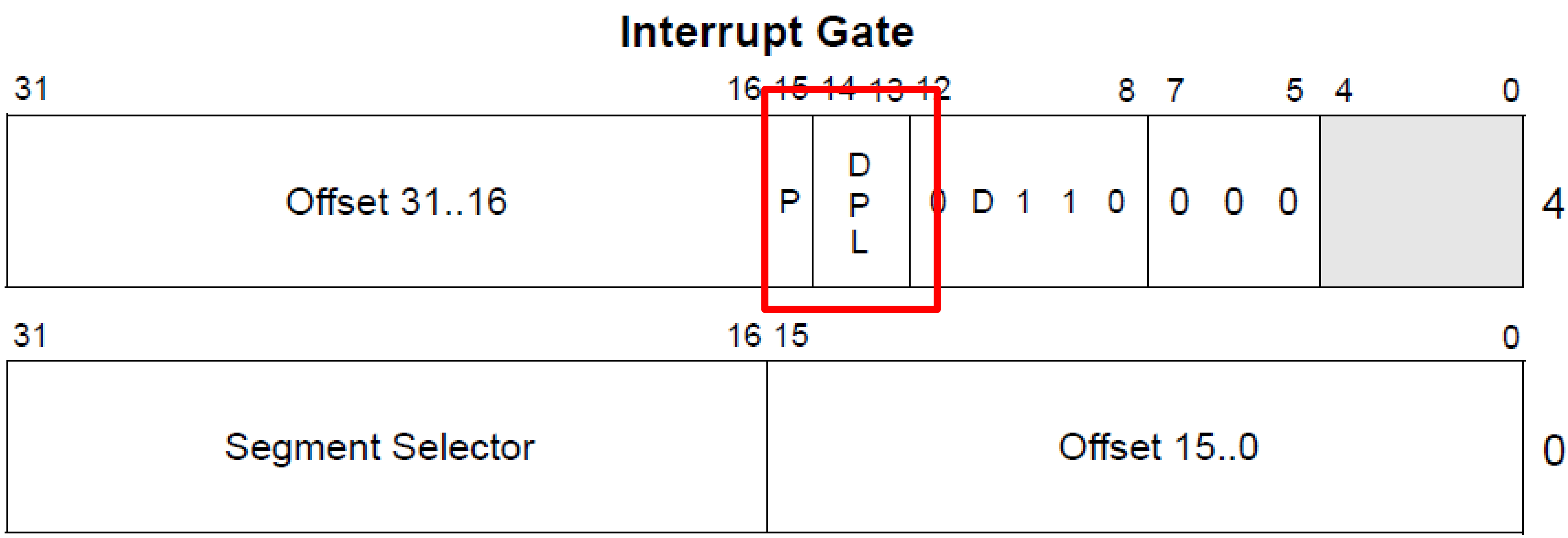
```
void
tvinit(void)
{
  int i;

  for(i = 0; i < 256; i++)
    SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
  SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3,
          vectors[T_SYSCALL], DPL_USER);

  initlock(&tickslock, "time");
}
```



Protection

- Generally user code cannot invoke int X
  - 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)

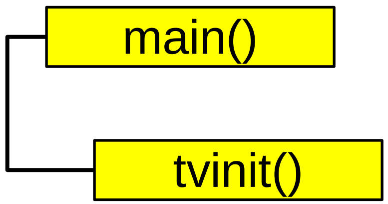


- Interrupt is allowed if...
  - current privilege level (CPL) is less or equal to descriptor privilege level (DPL)
- The kernel protects device interrupts from user

Initialize IDT

- A couple of important details

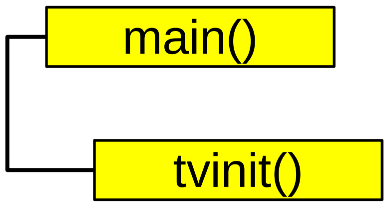
```
void
tvinit(void)
{
  int i;

  for(i = 0; i < 256; i++)
    SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
  SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3,
              vectors[T_SYSCALL], DPL_USER);

  initlock(&tickslock, "time");
}
```



Initialize IDT

- Only int T_SYSCALL can be called from user-level

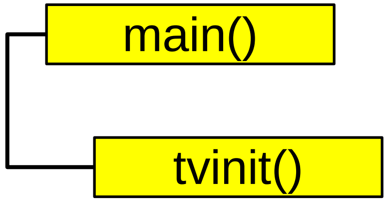
```
void
tvinit(void)
{
  int i;

  for(i = 0; i < 256; i++)
    SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
  SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3,
          vectors[T_SYSCALL], DPL_USER);

  initlock(&tickslock, "time");
}
```



Initialize IDT

```
void
tvinit(void)
{
  int i;

  for(i = 0; i < 256; i++)
    SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
  SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3,
          vectors[T_SYSCALL], DPL_USER);

  initlock(&tickslock, "time");
}
```

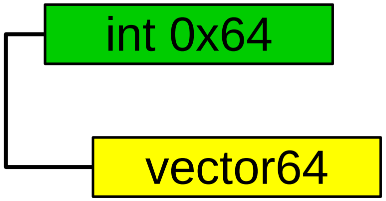
- Syscall is a “trap”
- i.e., doesn't disable interrupts



Interrupt path through the xv6 kernel

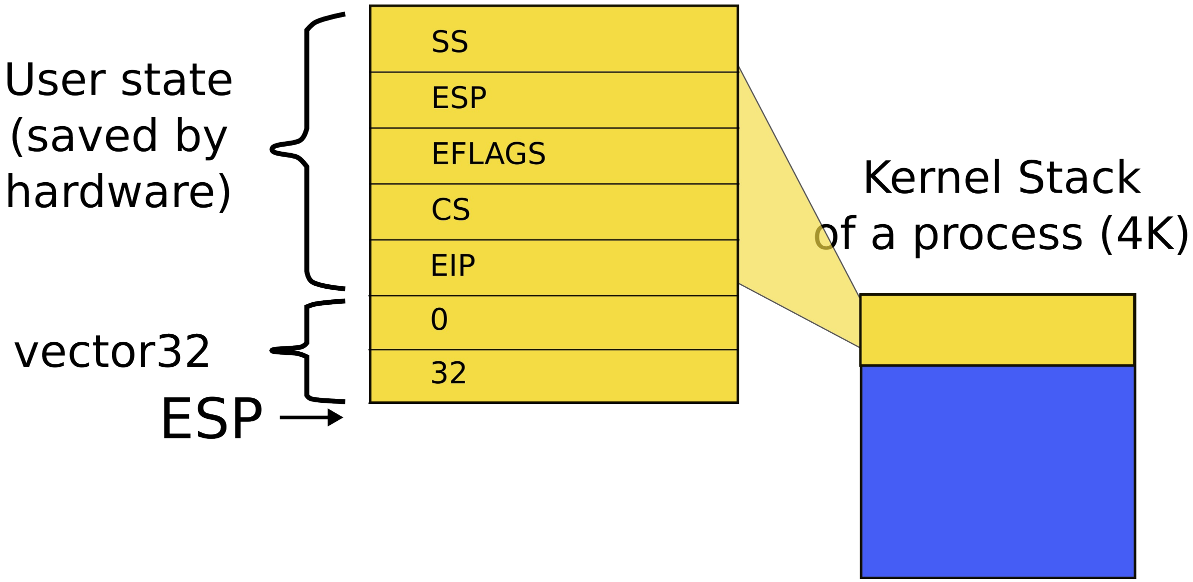
Where does IDT (entry 64) point to?

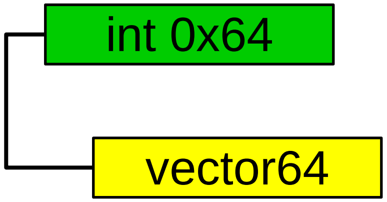
```
vector64:
  pushl $0    // error code
  pushl $64   // vector #
  jmp alltraps
```



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

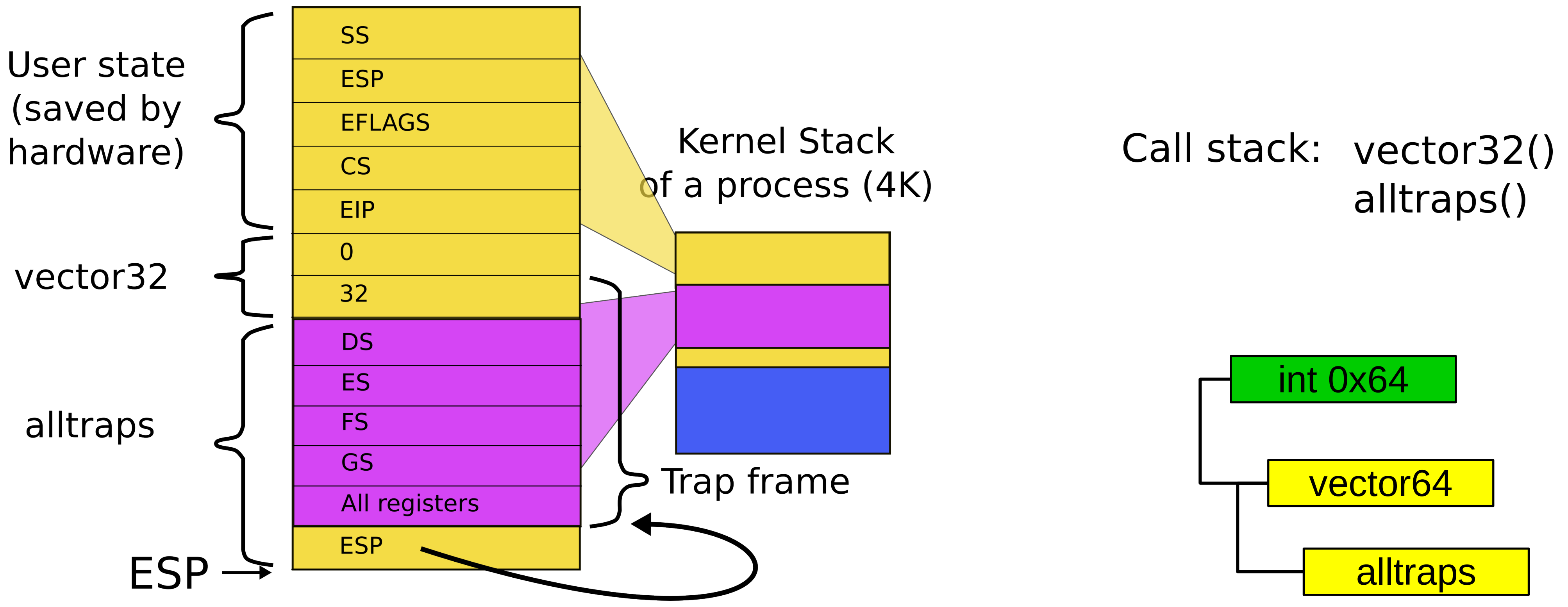
Kernel stack after interrupt





Call stack: vector32()

Kernel stack after interrupt

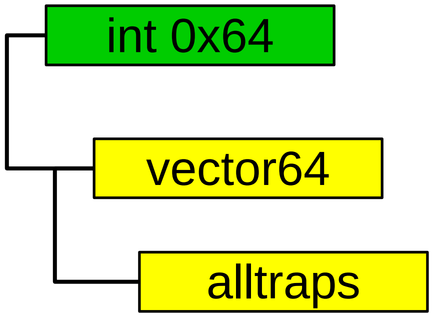


Syscall number

- 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

alltraps()

```
alltraps:
# Build trap frame.
pushl %ds
pushl %es
pushl %fs
pushl %gs
pushal

# Set up data and per-cpu segments.
movw $(SEG_KDATA<<3), %ax
movw %ax, %ds
movw %ax, %es
movw $(SEG_KCPU<<3), %ax
movw %ax, %fs
movw %ax, %gs

# Call trap(tf), where tf=%esp
pushl %esp
call trap
```



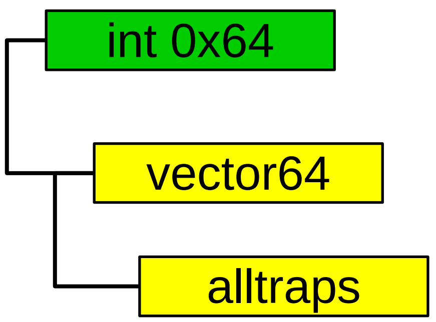
pusha

- An assembler instruction that saves all registers on the stack
  - 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);
```

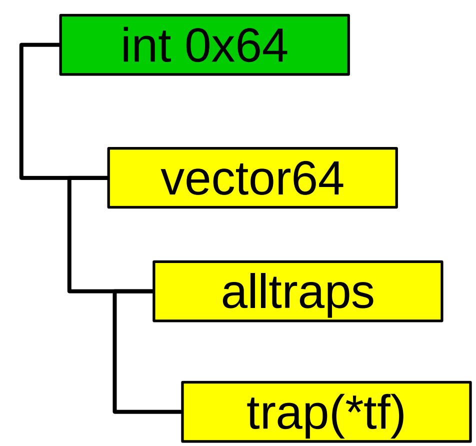
The end result: call trap()

```
alltraps:
# Build trap frame.
pushl %ds
pushl %es
pushl %fs
pushl %gs
pushal

# Set up data and per-cpu segments.
movw $(SEG_KDATA<<3), %ax
movw %ax, %ds
movw %ax, %es
movw $(SEG_KCPU<<3), %ax
movw %ax, %fs
movw %ax, %gs

# Call trap(tf), where tf=%esp
pushl %esp
call trap
```



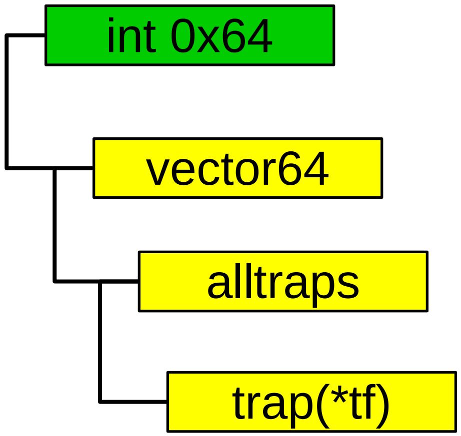
All interrupts, e.g. timer interrupt end up in a single funciton: trap()

```
trap(struct trapframe *tf)
{
...
  switch(tf->trapno){
  case T_IRQ0 + IRQ_TIMER:
    if(cpu->id == 0){
    acquire(&tickslock);
    ticks++;
    wakeup(&ticks);
    release(&tickslock);
  }
  break;
...
  if(proc && proc->state == RUNNING
      && tf->trapno == T_IRQ0+IRQ_TIMER)
    yield();
```



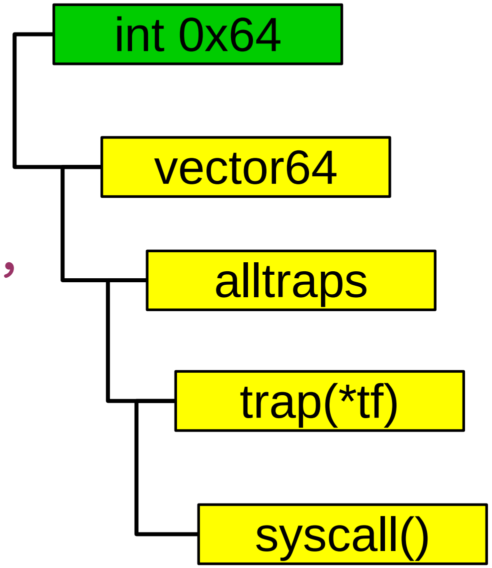
```
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(proc->killed)
      exit();
    proc->tf = tf;
    syscall();
    if(proc->killed)
      exit();
    return;
  }

  switch(tf->trapno){
  case T_IRQ0 + IRQ_TIMER:
```

Same for syscalls



syscall(): get the number from the trap frame

```
syscall(void)
{
  int num;

  num = proc->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num])
  {
    proc->tf->eax = syscalls[num]();
  } else {
    cprintf("%d %s: unknown sys call %d\n",
    proc->pid, proc->name, num);
    proc->tf->eax = -1;
  }
}
```



syscall(): process a syscall from the table

```
syscall(void)
{
  int num;

  num = proc->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num])
  {
     proc->tf->eax = syscalls[num]();
  } else {
    cprintf("%d %s: unknown sys call %d\n",
    proc->pid, proc->name, num);
    proc->tf->eax = -1;
  }
}
```

System call table

```
static int (*syscalls[])(void) = {
  [SYS_fork] sys_fork,
  [SYS_exit] sys_exit,
  [SYS_wait] sys_wait,
  [SYS_pipe] sys_pipe,
  [SYS_read] sys_read,
  [SYS_kill] sys_kill,
  [SYS_exec] sys_exec,
  [SYS_fstat] sys_fstat,
  [SYS_chdir] sys_chdir,
  [SYS_dup] sys_dup,
  [SYS_getpid] sys_getpid,
  [SYS_sbrk] sys_sbrk,
  [SYS_sleep] sys_sleep,
  [SYS_uptime] sys_uptime,
  [SYS_open] sys_open,
  [SYS_write] sys_write,
  [SYS_mknod] sys_mknod,
  [SYS_unlink] sys_unlink,
  [SYS_link] sys_link,
  [SYS_mkdir] sys_mkdir,
  [SYS_close] sys_close,
};
```

How do user programs access system calls?

- It would be weird to write

```
pushl $argv
pushl $init
pushl $0 // where caller pc would be
movl $SYS_exec, %eax
int $T_SYSCALL
```

  - … every time we want to invoke a system call
  - This is an example for the exec() system call

user.h

```
// 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 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`

usys.S

```
#include "syscall.h"
#include "traps.h"

#define SYSCALL(name) \
  .globl name; \
  name: \
    movl $SYS_ ## name, %eax; \
    int $T_SYSCALL; \
    ret

SYSCALL(fork)
SYSCALL(exit)
SYSCALL(wait)
SYSCALL(pipe)
SYSCALL(read)
```

- 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)`

```
int
sys_write(void)
{
  struct file *f;
  int n;
  char *p;

  if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
    return -1;
  return filewrite(f, p, n);
}
```

Example

- Write system call
- `if (write(1, buf, n) != n)`

```
int
sys_write(void)
{
  struct file *f;
  int n;
  char *p;

  if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
    return -1;
  return filewrite(f, p, n);
}
```

```
// Fetch the nth 32-bit system call argument.
int
argint(int n, int *ip)
{
    return fetchint(proc->tf->esp + 4 + 4*n, ip);
}
```

```
// Fetch the int at addr from the current process.
int
fetchint(uint addr, int *ip)
{
    if(addr >= proc->sz || addr+4 > proc->sz)
        return -1;
    *ip = *(int*)(addr);
    return 0;
}
```

argint(int n, int *ip)

```c
// Fetch the nth 32-bit system call argument.
int
argint(int n, int *ip)
{
    return fetchint(proc->tf->esp + 4 + 4*n, ip);
}
```

```c
// Fetch the int at addr from the current process.
int
fetchint(uint addr, int *ip)
{
    if(addr >= proc->sz || addr+4 > proc->sz)
        return -1;
    *ip = *(int*)(addr);
    return 0;
}
```

argint(int n, int *ip)

```
// Fetch the nth 32-bit system call argument.
int
argint(int n, int *ip)
{
    return fetchint(proc->tf->esp + 4 + 4*n, ip);
}
```

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

```
// Fetch the int at addr from the current process.
int
fetchint(uint addr, int *ip)
{
    if(addr >= proc->sz || addr+4 > proc->sz)
        return -1;
    *ip = *(int*)(addr);
    return 0;
}
```

argint(int n, int *ip)

```
// Fetch the nth 32-bit system call argument.
int
argint(int n, int *ip)
{
    return fetchint(proc->tf->esp + 4 + 4*n, ip);
}
```

- Skip return eip

```
// Fetch the int at addr from the current process.
int
fetchint(uint addr, int *ip)
{
    if(addr >= proc->sz || addr+4 > proc->sz)
        return -1;
    *ip = *(int*)(addr);
    return 0;
}
```

argint(int n, int *ip)

```c
// Fetch the nth 32-bit system call argument.
int
argint(int n, int *ip)
{
    return fetchint(proc->tf->esp + 4 + 4*n, ip);
}
```

- Fetch n'th argument

```c
// Fetch the int at addr from the current process.
int
fetchint(uint addr, int *ip)
{
    if(addr >= proc->sz || addr+4 > proc->sz)
        return -1;
    *ip = *(int*)(addr);
    return 0;
}
```

argint(int n, int *ip)

```c
// Fetch the nth 32-bit system call argument.
int
argint(int n, int *ip)
{
    return fetchint(proc->tf->esp + 4 + 4*n, ip);
}
```

```c
// Fetch the int at addr from the current process.
int
fetchint(uint addr, int *ip)
{
    if(addr >= proc->sz || addr+4 > proc->sz)
        return -1;
    *ip = *(int*)(addr);
    return 0;
}
```

fetchint(uint addr, int *ip)

```
// Fetch the nth 32-bit system call argument.
int
argint(int n, int *ip)
{
    return fetchint(proc->tf->esp + 4 + 4*n, ip);
}
```

```
// Fetch the int at addr from the current process.
int
fetchint(uint addr, int *ip)
{
    if(addr >= proc->sz || addr+4 > proc->sz)
        return -1;
    *ip = *(int*)(addr);
    return 0;
}
```

fetchint(uint addr, int *ip)

Any idea for what argptr() shall do?

- Write system call
- `if (write(1, buf, n) != n)`

```
int
sys_write(void)
{
  struct file *f;
  int n;
  char *p;

  if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
    return -1;
  return filewrite(f, p, n);
}
```

- Remember, buf is a pointer to a region of memory
  - i.e., a buffer
- of size n

```c
// Fetch the nth word-sized system call argument as a pointer
// to a block of memory of size n bytes. Check that the pointer
// lies within the process address space.
int
argptr(int n, char **pp, int size)
{
  int i;

  if(argint(n, &i) < 0)
    return -1;
  if((uint)i >= proc->sz || (uint)i+size > proc->sz)
    return -1;
  *pp = (char*)i;
  return 0;
}
```

- Check that the pointer to the buffer is sound

argptr(uint addr, int *ip)

```c
// Fetch the nth word-sized system call argument as a pointer
// to a block of memory of size n bytes. Check that the pointer
// lies within the process address space.
int
argptr(int n, char **pp, int size)
{
  int i;

  if(argint(n, &i) < 0)
    return -1;
  if((uint)i >= proc->sz || (uint)i+size > proc->sz)
    return -1;
  *pp = (char*)i;
  return 0;
}
```

- Check that the buffer is in user memory

argptr(uint addr, int *ip)

Summary

- We've learned how system calls work

Thank you