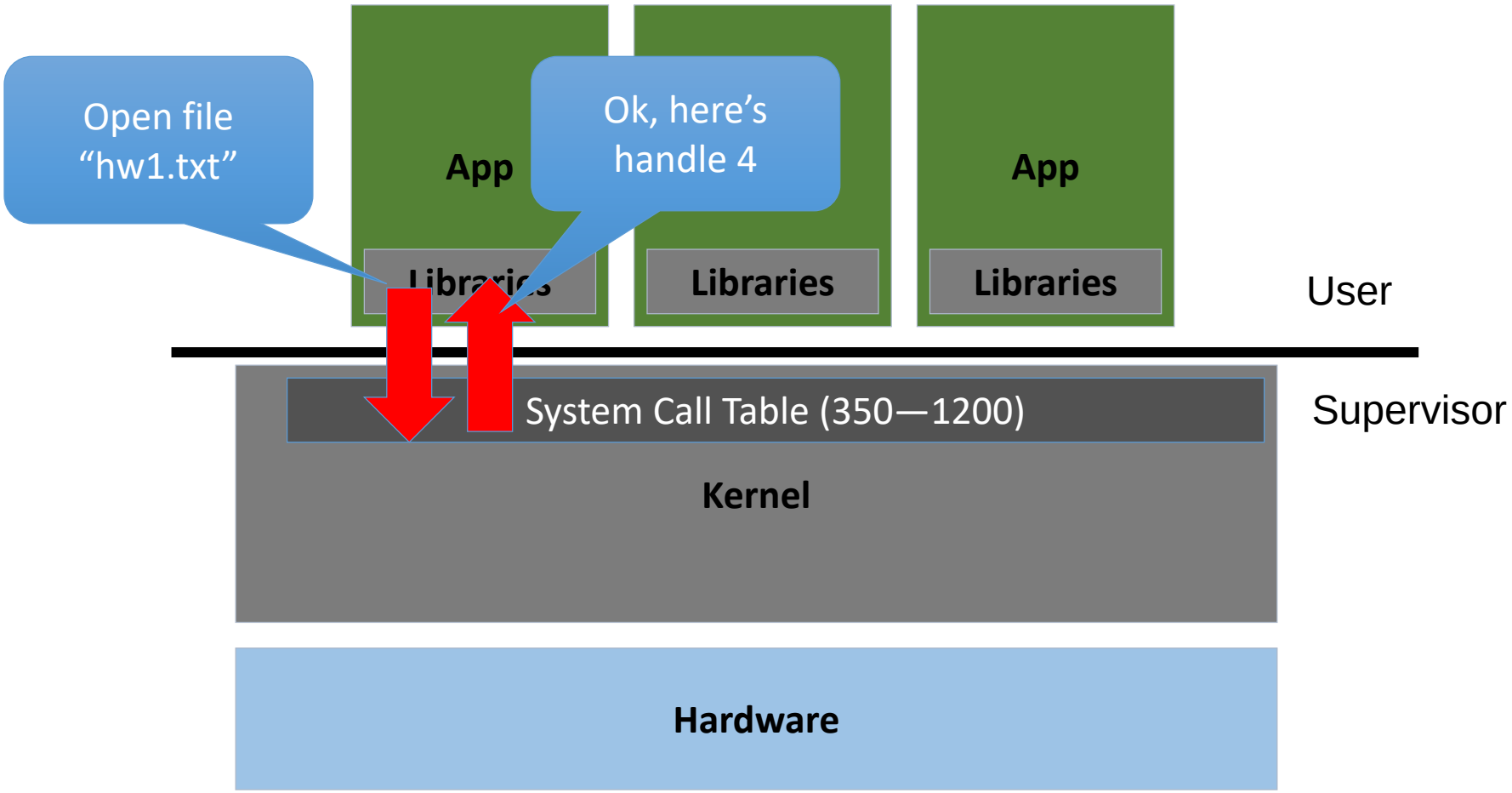
Interrupts & System Calls

Nima Honarmand

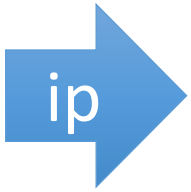
Previously on CSE306…



Regular Control Flow

- Regular instruction flow in a processor
  - Fetch the instruction pointed to by `ip` (instruction pointer) register
  - Execute the current instruction
  - Increment `ip` to point to the next instruction
    - If current inst is a jump, branch or call, set `ip` to its target instead of incrementing
- This is called regular control flow because the program itself determines the next instruction at any step
  - Instruction flow logically follows the course code

Regular Control Flow




```
x = 2, y = true

if (y) {

    x /= 2;

    printf(x);

} //...
```

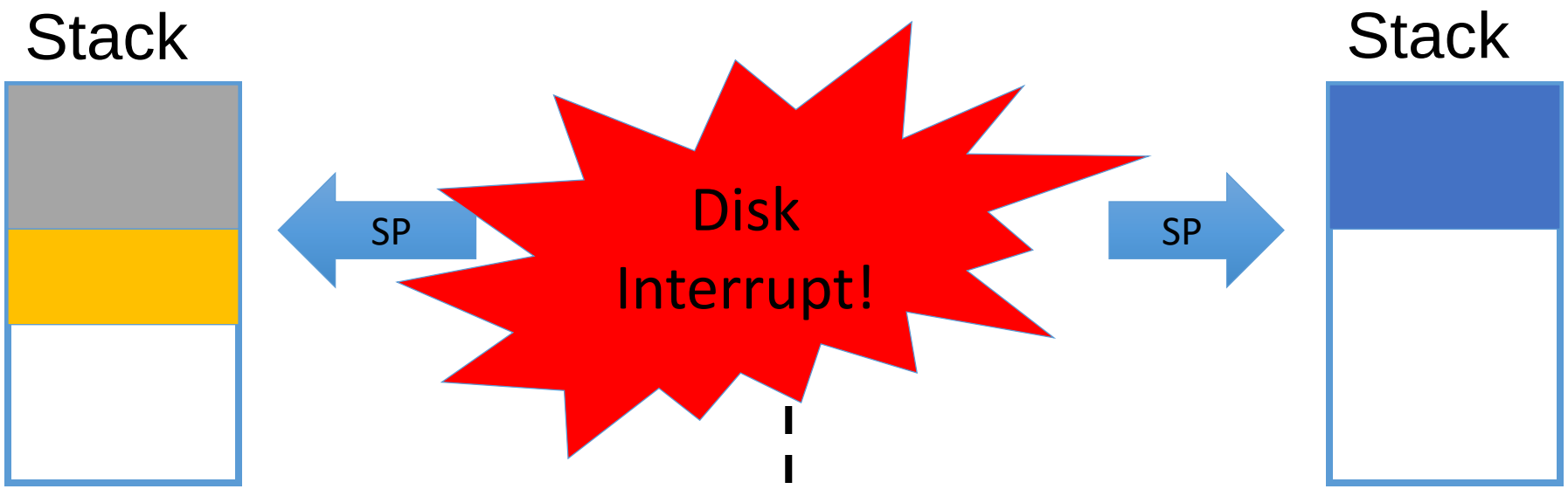
```
void printf(va_args)
{

    //...

}
```

Regular control flow: branches and calls (logically follows source code)

Irregular Control Flow

- Some times, due to “special events”, control has to be transferred to somewhere outside the program
- Since the program does not determine the target in this case, we call it irregular control flow
- Three cases in an OS:
  - External interrupt: caused by a hardware device, e.g., timer ticks, network card interrupts
  - Trap: Explicitly caused by the current execution, e.g., a system call
  - Exception (or Fault): Implicitly caused by the current execution, e.g., a page fault or a device-by-zero fault

External Interrupt Example



How to Handle?

- Five general steps
  1) Transfer control to a pre-specified instruction in the kernel code
     - Who should specify this location?
  2) Save current thread's "context" on the kernel stack
     - Why?
  3) Execute a service routine to handle the situation
  4) Restore the current thread context
  5) Return to the interrupted code, right after the last executed instruction

How to Handle?

- External interrupts, traps and exceptions can all use the same five-step procedure
- So they do: Intel provides a single mechanism to handle all of them
- We use the general term **interrupt** to refer to all of them, unless stated otherwise

How it works: Hardware

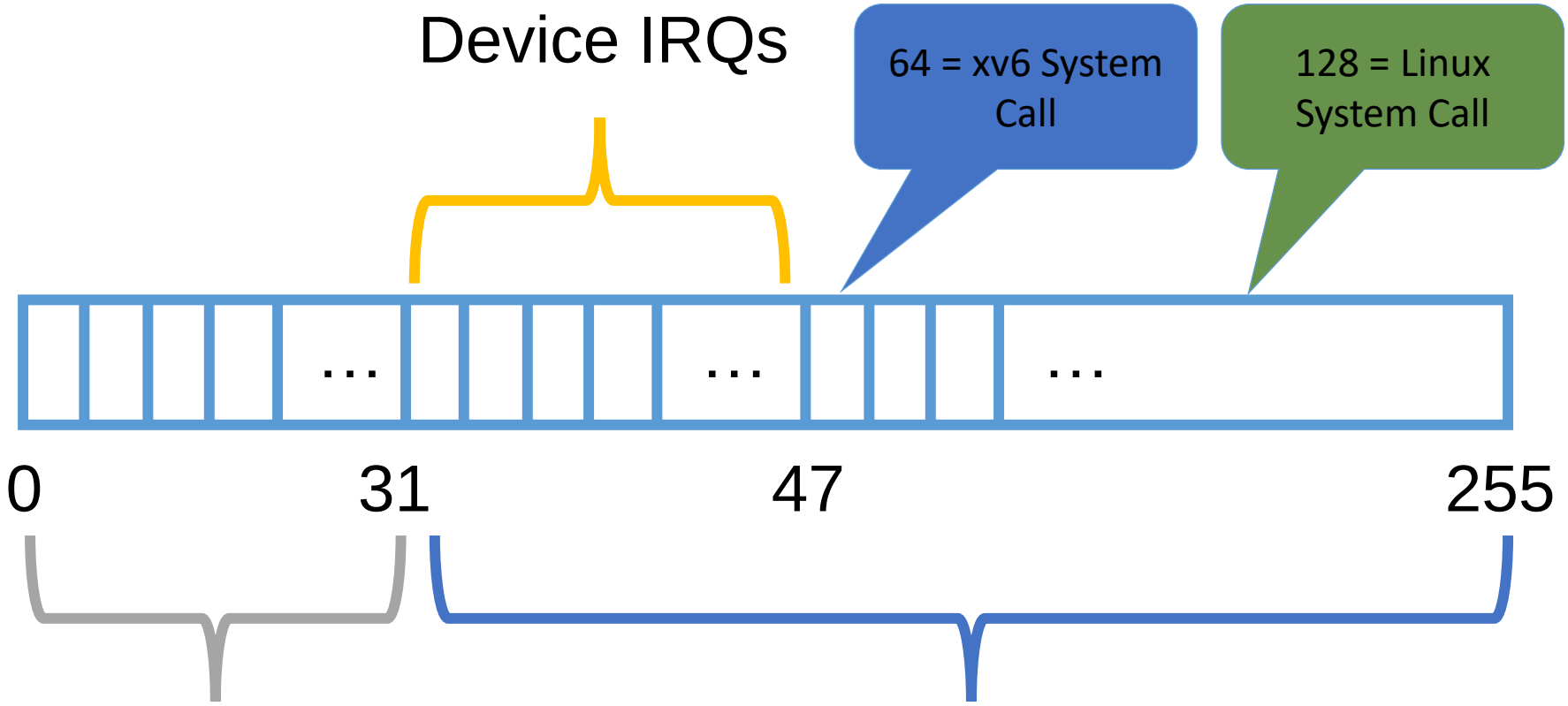
Interrupt Number (Vector)

- Each interrupt identified a number indicating its type
- E.g., in x86, 14 is a page fault, 3 is a debug breakpoint
- This number is the index into an Interrupt Descriptor Table (IDT) stored in memory

x86 Interrupt Overview

- Support 256 interrupts (assigned an index from 0-255)
- #0-31 are for processor interrupts; generally fixed by Intel
  - E.g., 14 is always for page faults
- 32-255 are software configured
  - 32-47 are for device interrupts (IRQs) in xv6
    - Most device's IRQ line can be configured
    - Look Chapter 4 of Bovet and Cesati for more details
  - xv6 uses #64 (0x40) for its system call
  - Linux uses #128 (0x80) for its system call

x86/xv6 Interrupts

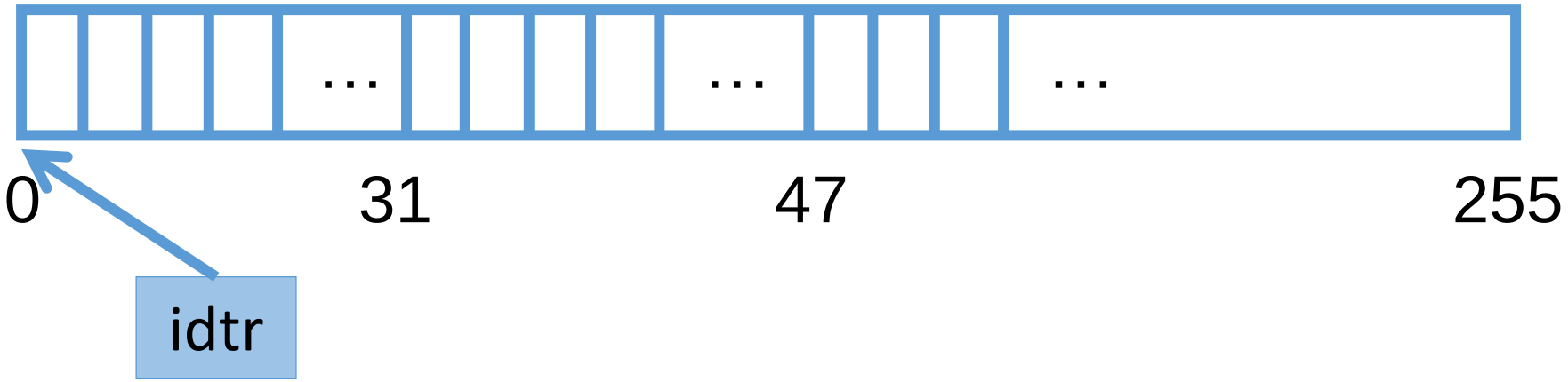


Traps (Software Interrupts)

- In x86, the `int <num>` instruction allows software to raise an interrupt
  - So in an xv6 user-mode program, if you see `int 0x40`, it's a system call
- OS sets ring level required to raise an interrupt
  - Generally, user programs can't manually issue an `int 14` (page fault)
  - An unauthorized `int` instruction causes a General Protection (#GP) fault
    - Interrupt #13

How Is This Configured?

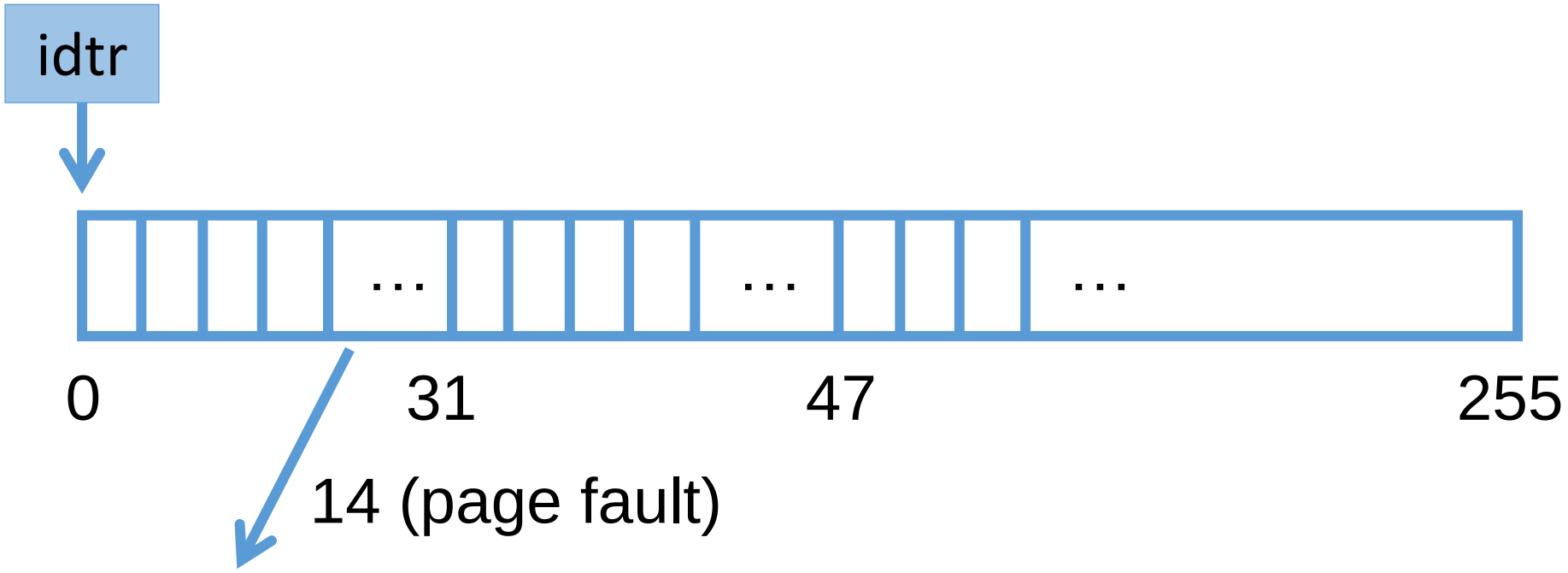
- Kernel creates an array of Interrupt Descriptors in memory, called Interrupt Descriptor Table, or IDT
  - Can be anywhere in memory
  - Pointed to by special processor register (`idtr`)
- Entry 0 configures interrupt 0, and so on



Interrupt Descriptor

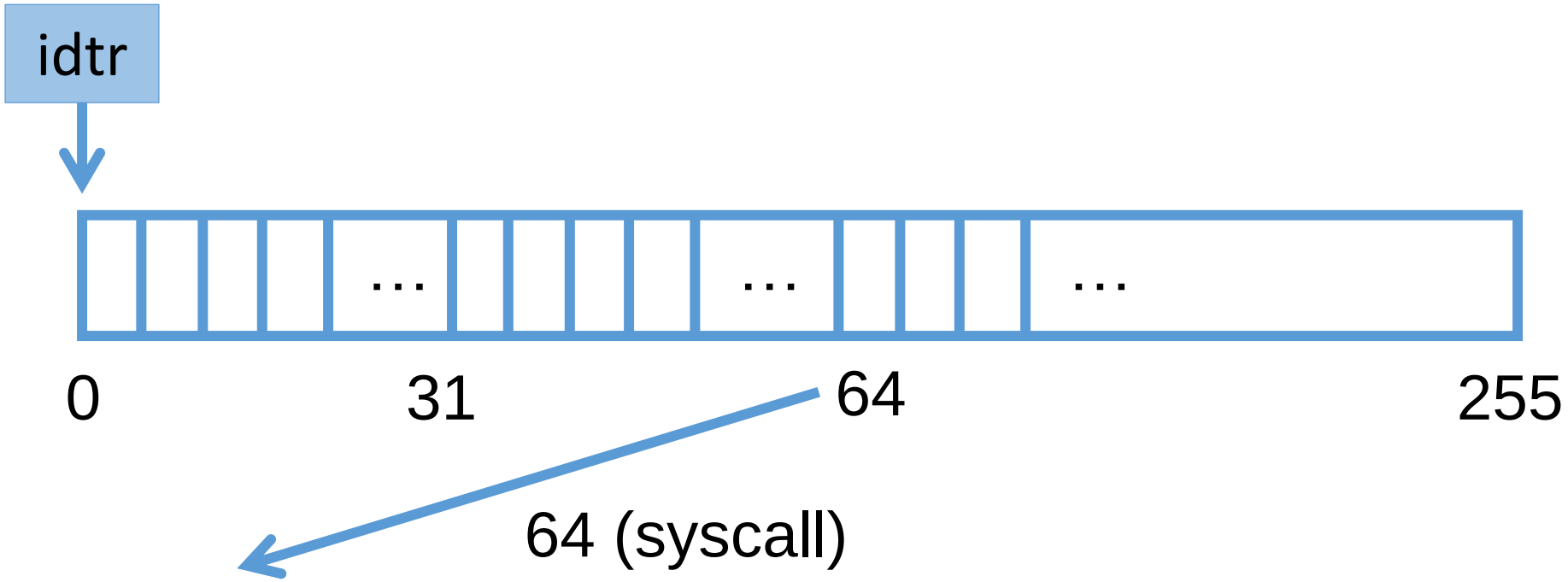
- Code segment selector
  - Almost always the same (kernel code segment)
- Address of the code to run
- Privilege Level (Ring)
  - What is the minimum privilege level that can invoke the interrupt (using `int` instruction)
- Present bit – disable unused interrupts
- And a bunch of other stuff…

IDT Example: Page Fault




```
Code Segment: Kernel Code
Segment Offset: &page_fault_handler
Ring: 0 // user code may not raise this exception
Present: 1
```

IDT Example: xv6 Syscall




```
Code Segment: Kernel Code
Segment Offset: &syscall_handler
Ring: 3 // user code may raise this exception
Present: 1
```

x86 Interrupt Descriptors

- x86 interrupt descriptors support many other (legacy) features that are rarely used
- Makes their working and in-memory layout a bit confusing
- Look at the architecture manual for more details

xv6 code review

- Five general steps
  1) Transfer control to a pre-specified instruction in the kernel code
  2) Save current thread's "context" on the kernel stack
  3) Execute a service routine to handle the situation
  4) Restore the current thread context
  5) Return to the interrupted code, right after the last executed instruction

Read the xv6-book (chapter 3) for a detailed review.

How it works: Software

xv6 code review

- Five general steps
  1) Transfer control to a pre-specified instruction in the kernel code
  2) Save current thread's "context" on the kernel stack
  3) Execute a service routine to handle the situation
  4) Restore the current thread context
  5) Return to the interrupted code, right after the last executed instruction

Read the xv6-book (chapter 3) for a detailed review.

System Calls

System Call "Interrupt"

- System calls issued using `int` instruction
  - `int 0x40` in xv6
  - `int 0x80` in Linux
- Dispatch routine is just an interrupt handler
- System calls are arranged in a table
  - See `syscall.h` and `syscall.c` in xv6
- Program selects the one it wants by placing index in `eax` register before executing the `int` instruction
  - Arguments go in the other registers or on the stack, as specified by the OS
  - Return value goes in `eax`

How many system calls?

- Linux exports about 350 system calls
- Windows exports about 400 system calls for core APIs, and another 800 for GUI methods

xv6 code review

- System call table
- Remember, you will add your very own system call in Lab 1!

Again, Read the xv6-book (chapter 3) for a detailed review.

New System Call Instructions (1)

Around Pentium 4 era (2000):

- Processors got very deeply pipelined
  - Pipeline stalls/flushes became very expensive
  - Cache misses can cause pipeline stalls
- System calls took twice as long from Pentium 3 to Pentium 4
  - Why?
  - IDT entry may not be in the cache
  - Different permissions constrain instruction reordering

New System Call Instructions (2)

- Idea: what if we cache the IDT entry for a system call in a special CPU register?
  - No more cache misses for the IDT!
  - Maybe we can also do more optimizations
- Assumption: system calls are frequent enough to be worth the transistor budget to implement this

AMD: `syscall` & `sysret`

- These instructions uses an MSR (machine specific registers) to store syscall entry point and code segment
- A drop-in replacement for `int 0x80`
- Everyone loved it and adopted it wholesale
  - Even Intel!
- Intel later added its own instructions
  - `sysenter` and `sysexit`