Interrupts & System Calls

Nima Honarmand

What is an Interrupt?

- Loosely defined: an *irregular* control-flow from one context of execution and back
  - Usually, user to kernel and back, can also happen within kernel
- Types of interrupts:
  - 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: Implicitly caused by the current execution, e.g., a page fault or a device-by-zero fault
- External interrupts are asynchronous interrupts
  - Not caused by the last instruction executed
- Traps and exceptions are synchronous interrupts
  - Caused by the last instruction executed

How to Handle Interrupts

1. Save current execution context
   - Why?
   - Because it's irregular control flow, so program cannot save its own state
2. Transfer control to a well-defined location in the kernel code
   - Switch privilege levels as needed
3. Handle the interrupt
4. Return to the previous context after handling the interrupt
   - Should restore the saved state

→ It seems all three forms of interrupts can use the same general mechanism
   - That's why we discuss them together

Interrupt Handling (Hardware)

What Happens (Generally):

- Control jumps to the kernel
  - At a prescribed address (the interrupt handler)
- The register state of the program is dumped on the kernel's stack
  - Sometimes, extra info is loaded into CPU registers
  - E.g., page faults store the address that caused the fault in the `cr2` register
- Kernel code runs and handles the interrupt
- When handler completes, resume program (see `iret` instruction)

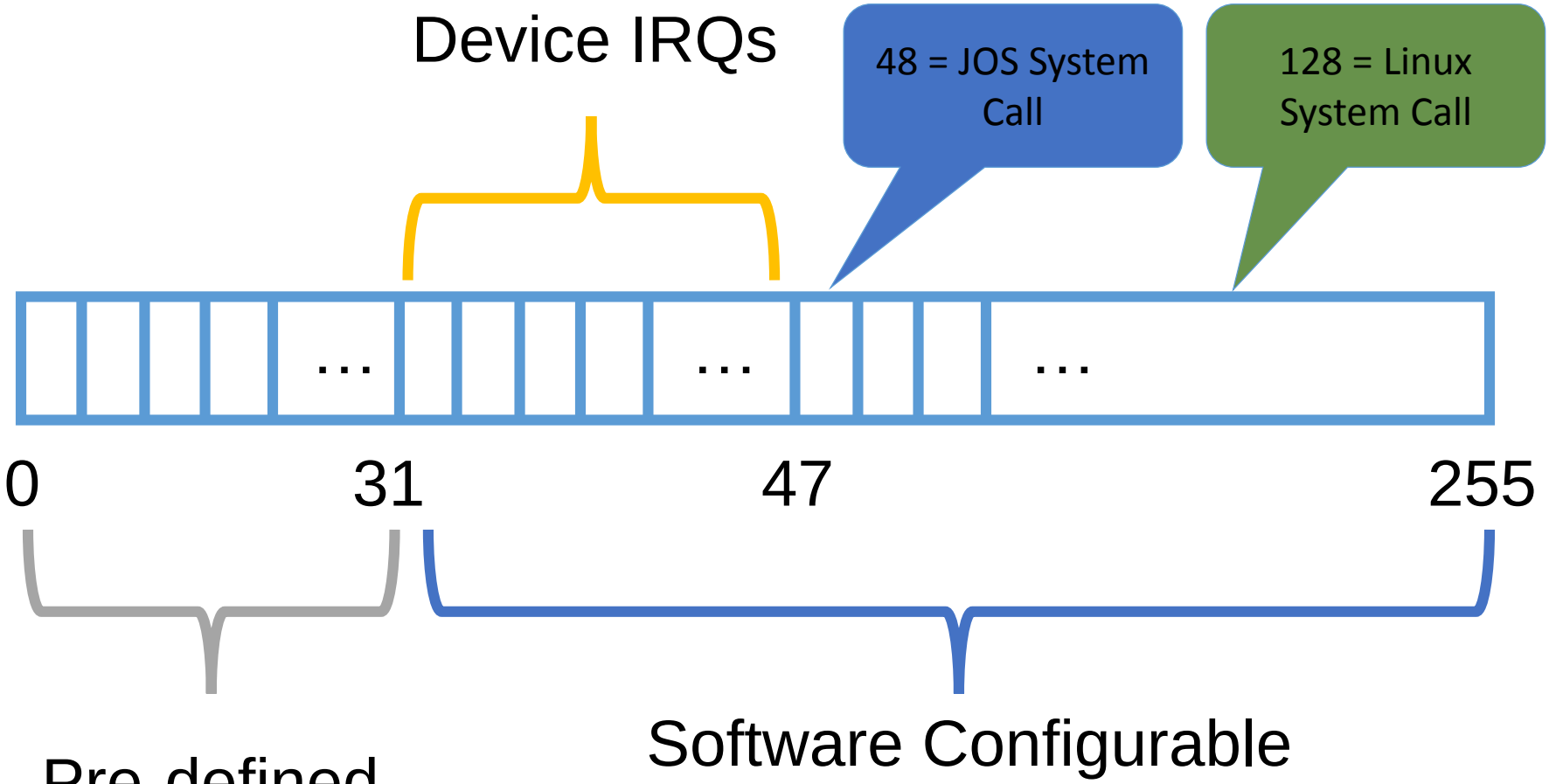
How Does It Work?

- How does HW know what to execute?
- Where does the HW dump the registers; what does it use as the interrupt handler's stack?

Interrupt Overview

- Each external interrupt, exception or trap includes a number indicating its type
  - From now on, interrupt means “external interrupt, exception or trap” unless otherwise specified
- E.g., 14 is a page fault, 3 is a debug breakpoint
- This number is the index into an Interrupt Descriptor Table

x86 Interrupts



x86 Interrupt Overview

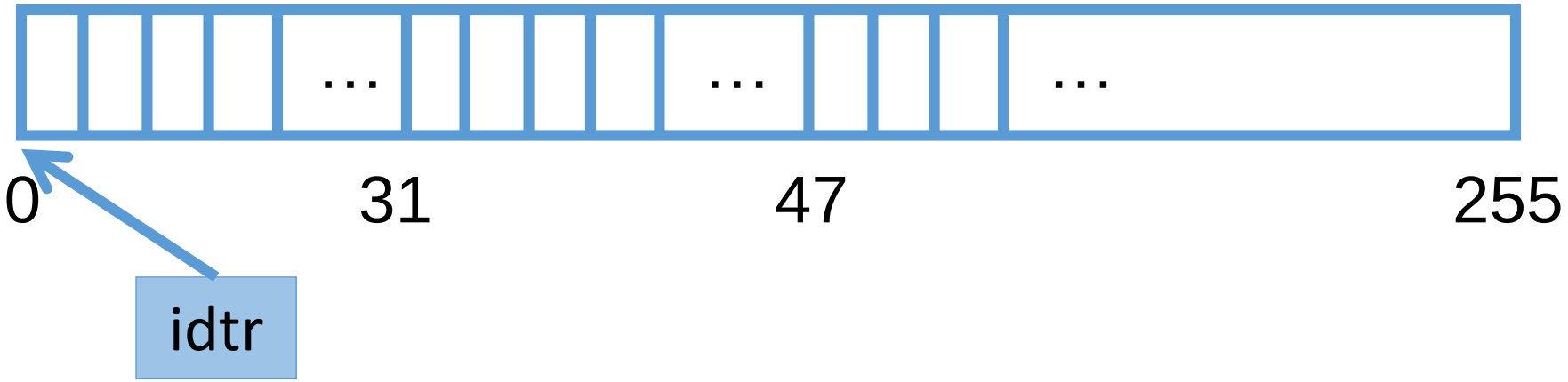
- Each interrupt is assigned an index from 0-255
- 0-31 are for processor interrupts; generally fixed by Intel
  - E.g., 14 is always for page faults
- 32-255 are software configured
  - 32-47 are for device interrupts (IRQs) in JOS
    - Most device's IRQ line can be configured
    - Look up APICs for more info (Chapter 4 of Bovet and Cesati)
  - 128 (0x80) and 48 (0x30) issue system calls in Linux and JOS respectively

Software Interrupts

- The `int <num>` instruction allows software to raise an interrupt
  - 0x80 is just a Linux convention. JOS uses 0x30.
- OS sets ring level required to raise an interrupt
  - Generally, user programs can't issue an `int 14` (page fault manually)
  - An unauthorized `int` instruction causes a General Protection (#GP) fault
    - Interrupt 13

How Is This Configured?

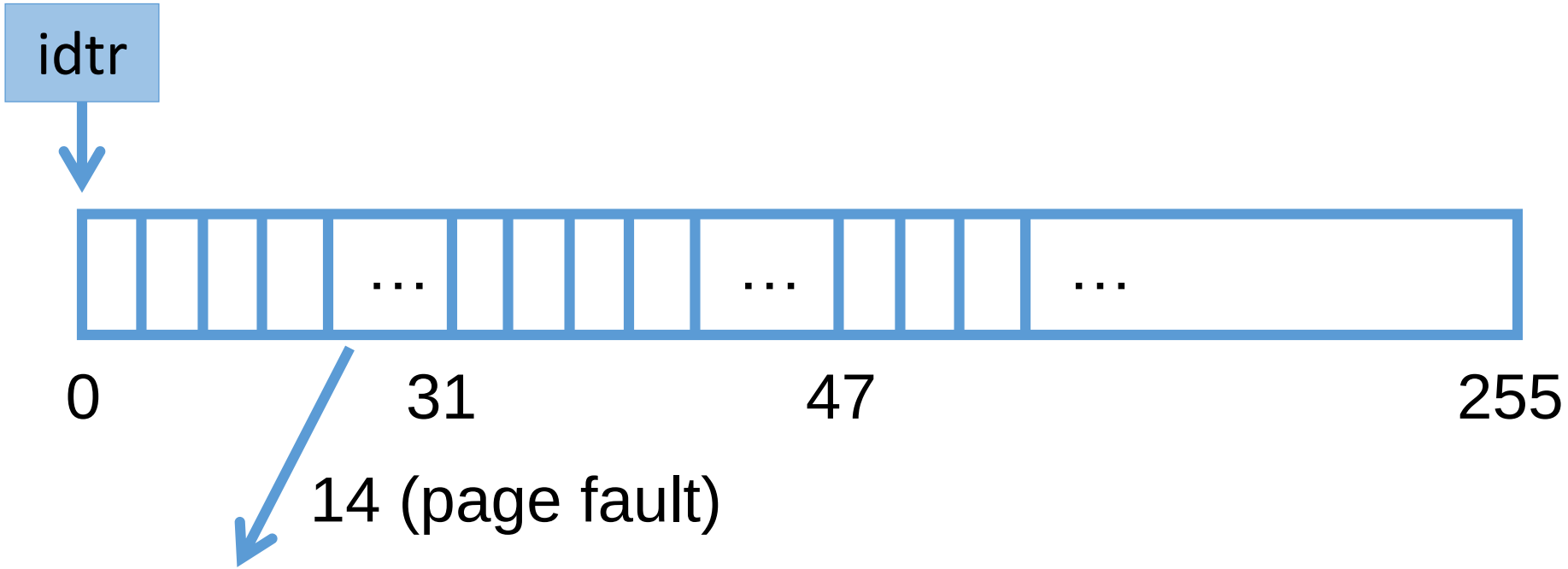
- Kernel creates an array of Interrupt descriptors in memory, called Interrupt Descriptor Table, or IDT
  - Can be anywhere in memory
  - Pointed to by special register (`idtr`)
    - c.f., segment registers and `gdtr` and `ldtr`
- Entry 0 configures interrupt 0, and so on



Interrupt Descriptor

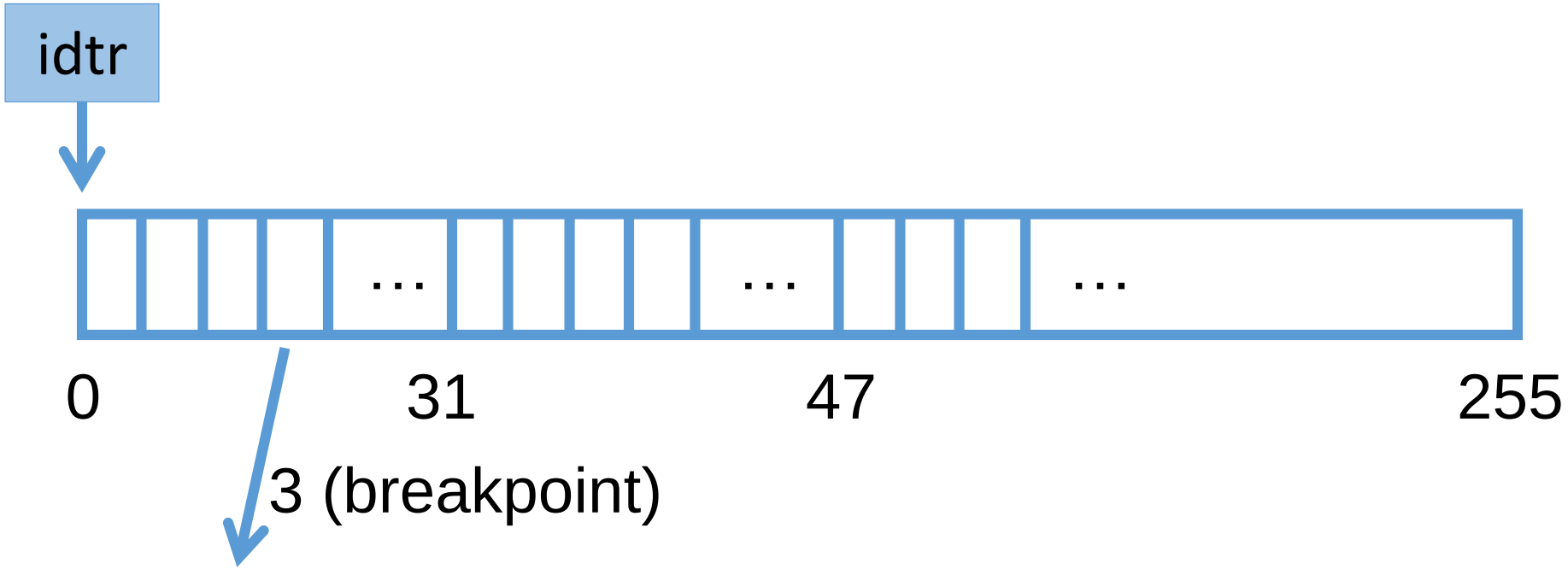
- Code segment selector
  - Almost always the same (kernel code segment)
- Segment offset of the code to run
  - If kernel segment is “flat” (Linux, JOS), this is just the linear address
- Privilege Level (Ring)
  - What is the minimum privilege level that can invoke the interrupt (using `int` instruction)
- Present bit – disable unused interrupts
- Gate type (interrupt or trap/exception) – more in a bit

x86 IDT Example: Page Fault




```
Code Segment: Kernel Code
Segment Offset: &page_fault_handler //linear addr
Ring: 0 // kernel
Present: 1
Gate Type: Exception
```

x86 IDT Example: Breakpoint




```
Code Segment: Kernel Code
Segment Offset: &breakpoint_handler //linear addr
Ring: 3 // user
Present: 1
Gate Type: Exception
```

Interrupt Descriptors (ctd)

- 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

How Does It Work?

- How does HW know what to execute?
  - Interrupt descriptor specifies what code to run and at what privilege
  - This can be set up once during boot for the whole system
- Where does the HW dump the registers; what does it use as the interrupt handler's stack?
  - Specified in the Task State Segment

Task State Segment (TSS)

- Another segment, just like the code and data segment
  - A descriptor created in the GDT (cannot be in LDT)
  - Selected by special task register (tr)
  - Unlike others, the segment content has a hardware-specified layout
- Lots of fields for rarely-used features
- Two features we care about in a modern OS:
  1. Location of kernel stack (fields ss0/esp0)
  2. I/O Port privileges (more in a later lecture)

TSS and Kernel Stack

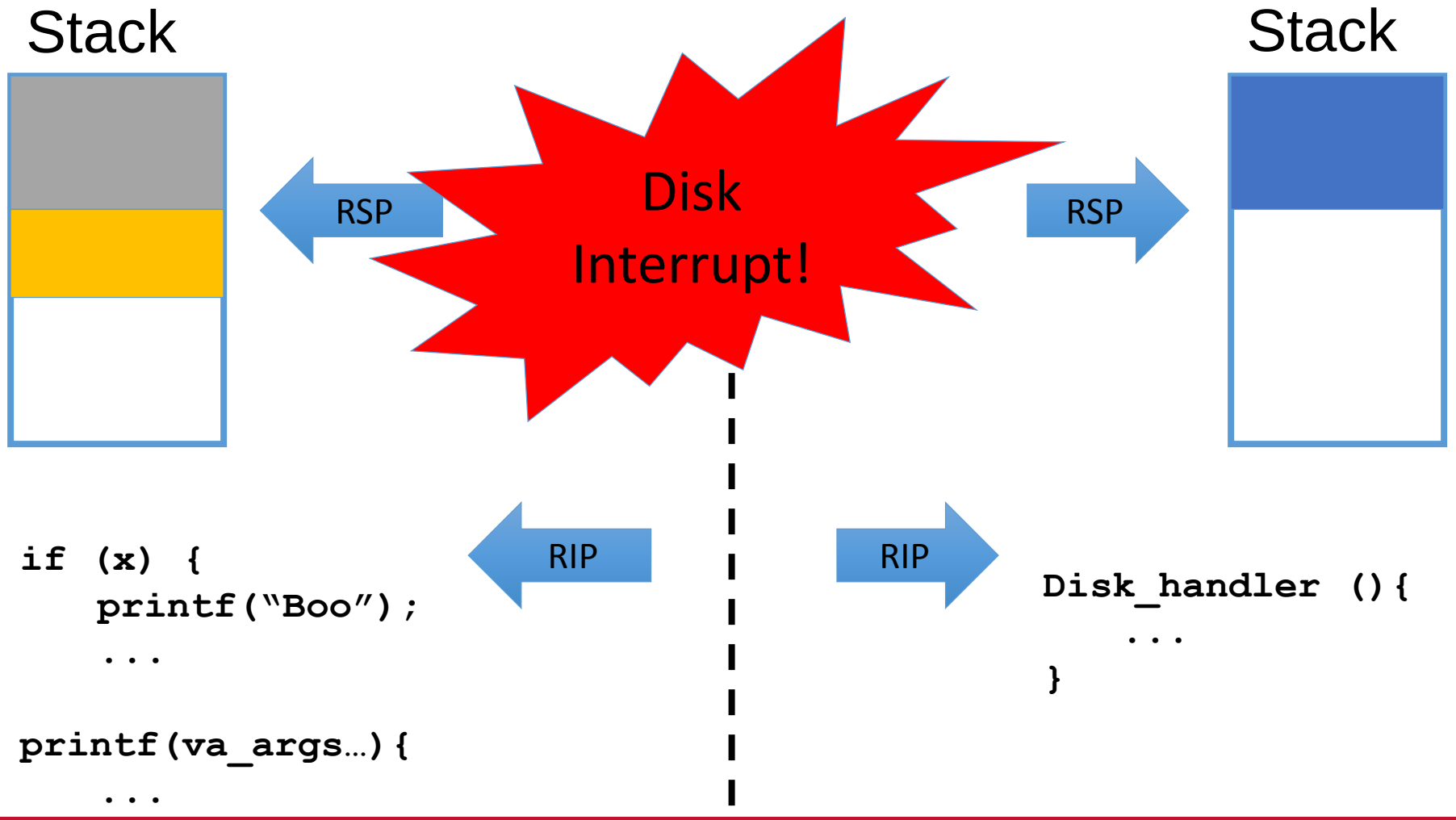
- Simple model: separate TSS segments and kernel stacks for each process
- Optimization (JOS):
  - Our kernel is pretty simple
  - Why not just share one TSS and kernel stack per-processor?
- Linux model:
  - Separate kernel stacks per process
  - One TSS per CPU
  - Modify TSS fields as part of context switching

Interrupt Handling (Software)

Interrupt Handling

- For now, just consider external interrupts

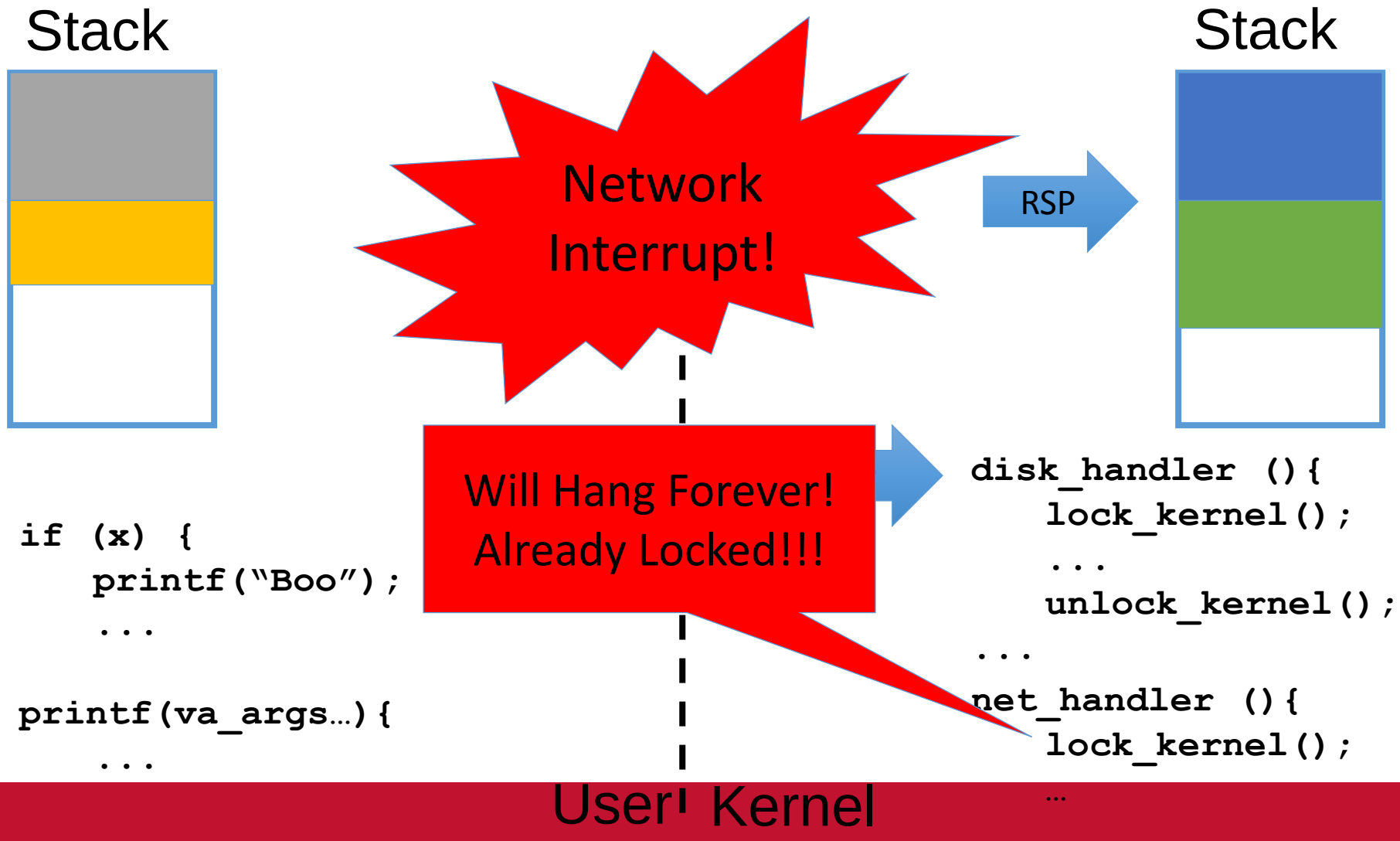
Example


Stack
Stack
RSP
Disk Interrupt!
RSP
RIP
RIP
if (x) {
printf("Boo");
...
printf(va_args…){
...
Disk_handler (){
...
}
User
Kernel

Complication:

- What happens if I'm in an interrupt handler, and another interrupt comes in?
- What could go wrong?
  - Hint: this creates some sort of concurrency
  - Violate code invariants (inconsistency) if not using locks
  - Deadlock if using locks
  - Exhaust the kernel stack (if too many fire at once)
    - kernel stack only changes on privilege level change; nested interrupts just push the next frame on the stack

Example


Stack
Stack
Network Interrupt!
RSP
Will Hang Forever! Already Locked!!!
if (x) {
printf("Boo");
...
printf(va_args…){
...
disk_handler (){
lock_kernel();
...
unlock_kernel();
...
net_handler (){
lock_kernel();
…
User
Kernel

Two Solutions

1. Make interrupt service routines (ISR) reentrant and synchronized
   - Difficult to get right
2. Disable further interrupts while serving current interrupt
   - Need to make ISR small and quick
     - Why?
     - Because devices often need quick attention
   - How?
     - By breaking interrupt handlers into ***Top*** and ***Bottom*** halves

Top & Bottom Halves

- Top Half: just acknowledge the interrupt, but postpone the actual work by adding a "work item" to some "work queue"
- Bottom Half: Do the actual processing later, after enabling interrupts, by traversing the work queue
- Only disable interrupts during the top half

Disabling Interrupts in x86

- An x86 CPU can disable I/O interrupts
  - Clear bit 9 of the EFLAGS register (IF Flag)
  - `cli` and `sti` instructions clear and set this flag

Gate types

- Recall: an IDT entry can be an interrupt or an exception gate
- Difference?
  - An interrupt gate automatically disables all other interrupts (i.e., clears IF on entry)
  - An exception gate doesn't

What about Exceptions?

- You can't mask or disable exceptions
  - Why not?
    - Can't make progress after a divide-by-zero
- Nested exceptions: do exception handlers need to be reentrant?
  - Not if your kernel has no bugs (or system calls in itself)
  - In certain cases, Linux allows nested page faults
    - E.g., to detect errors copying user-provided buffers
- Double and Triple faults
  - Exceptions encountered when hardware (not ISR) is trying to handle another interrupt/exception/trap
  - Example?

System Calls

System Call "Interrupt"

- Originally, system calls issued using `int` instruction
  - `int 0x80` in Linux
  - `int 0x30` in JOS
- Dispatch routine was just an interrupt handler
- Like interrupts, system calls are arranged in a table
  - See arch/x86/kernel/syscall_table*.S in Linux source
- Program selects the one it wants by placing index in `eax` register
  - Arguments go in the other registers by calling convention
  - Return value goes in `eax`

New System Call Instructions (1)

Around Pentium 4 era:

- Processors got very deeply pipelined
  - Pipeline stalls/flushes became very expensive
  - Cache misses can cause pipeline stalls
- System calls took twice as long from P3 to P4
  - Why?
  - IDT entry may not be in the cache
  - Different permissions constrain instruction reordering

New System Call Instructions (2)

- 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
  - What else could you do with extra transistors that helps performance?

AMD: `syscall` & `sysret`

- These instructions use MSRs (machine specific registers) to store:
  - Syscall entry point and code segment
  - Kernel stack
- 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`

In JOS

- You will use the `int` instruction to implement system calls
- There is a challenge problem in lab 3 to use `systenter/sysexit`
  - Note that there are some more details about register saving to deal with
  - `syscall/sysret` is a bit too trivial for extra credit
    - But still cool if you get it working!

VDSO

- Virtual Dynamic Shared Object
- A small shared library mapped automatically (by kernel) into the address space of processes
- Used to reduce the cost of some frequent system calls even further
  - E.g., `gettimeofday()`
- Done by mapping the data needed to serve the system call and the code to access that data into the process address space
- This way, the system call becomes a simple function call
  - no need to save/restore registers, switch PL, etc.