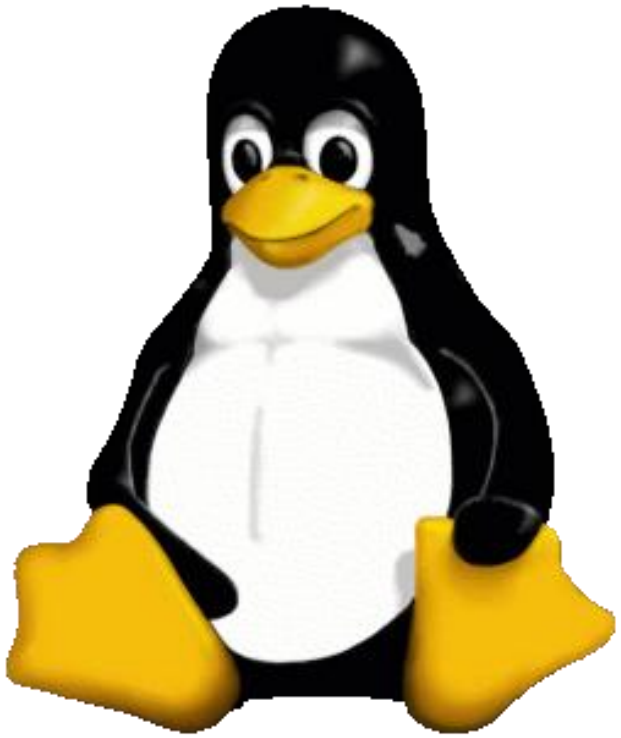
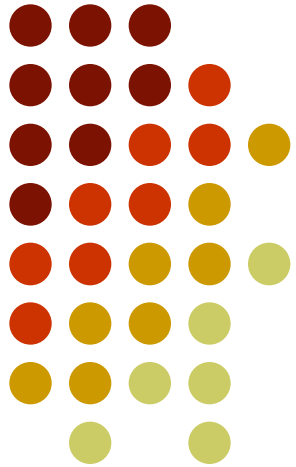
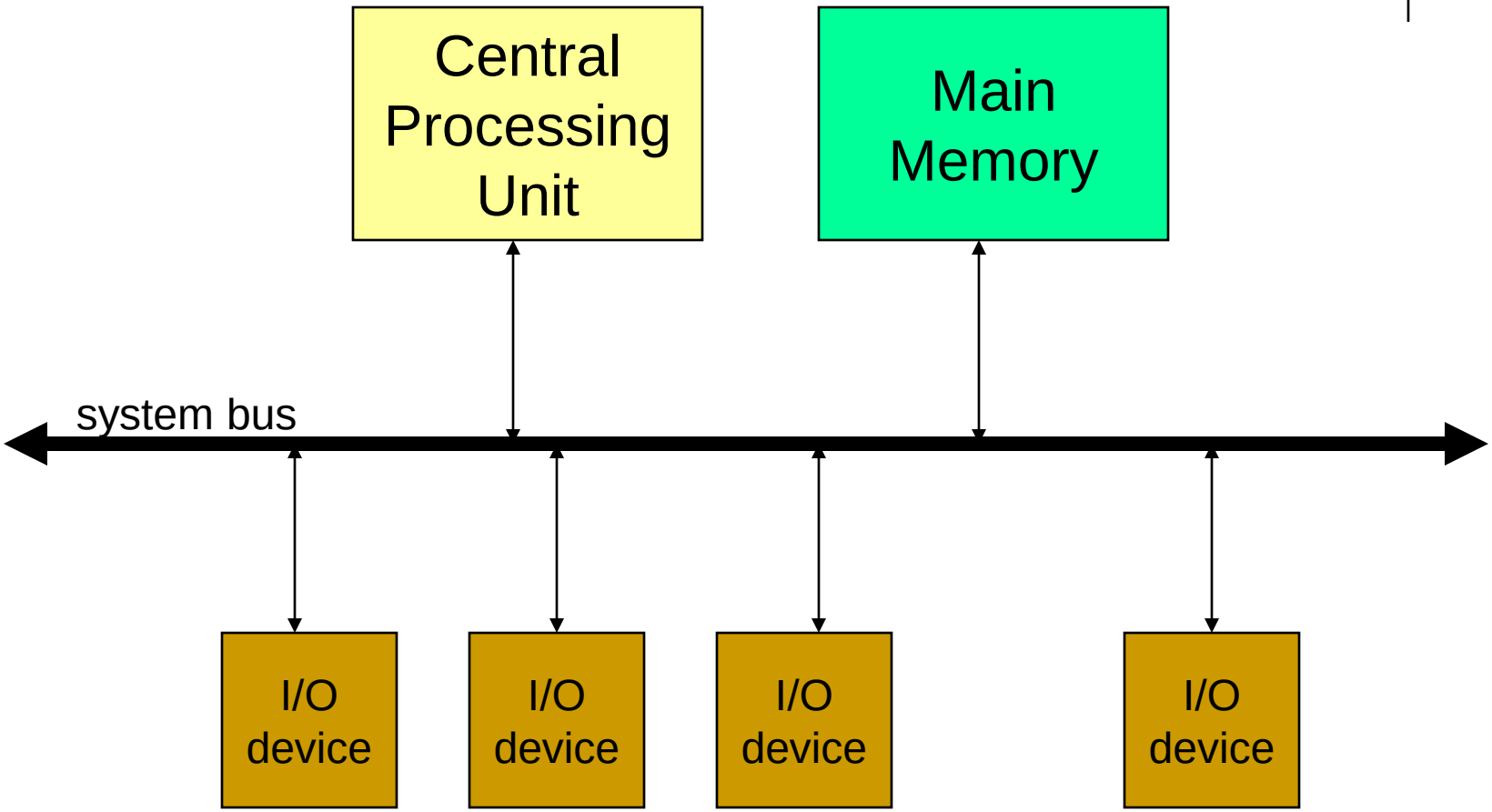
Programmed and Interrupts and Exceptions





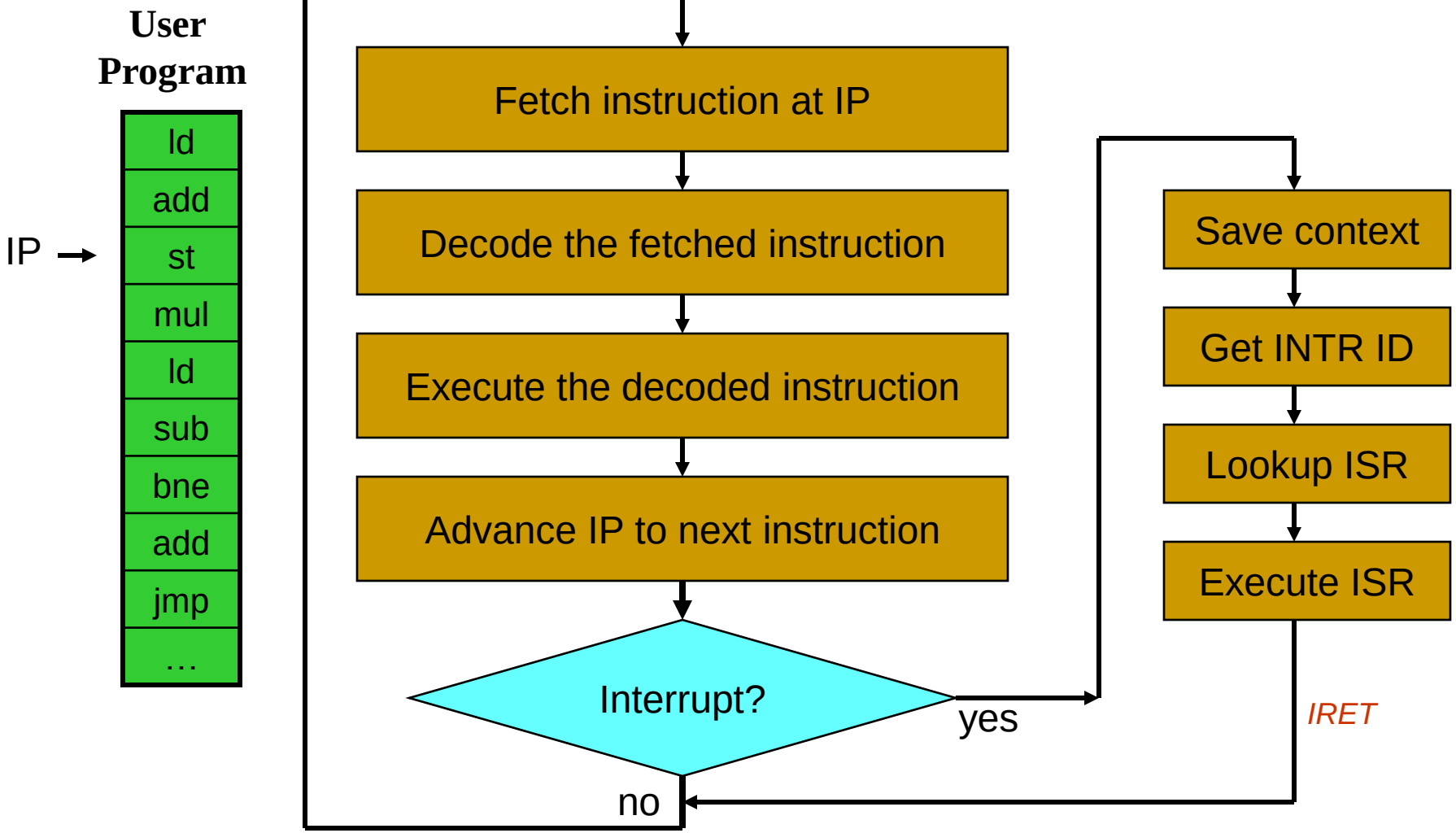
Simplified Architecture Diagram



Motivation

- Utility of a general-purpose computer depends on its ability to interact with I/O devices attached to it (e.g., keyboard, display, disk-drives, network, etc.)
- Devices require a prompt response from the CPU when various events occur, even when the CPU is busy running a program
- Need a mechanism for a device to “gain CPU’s attention”
- *Interrupts* provide a way doing this

CPU's 'fetch-execute' cycle



Interrupts

- Forcibly change normal flow of control
- Similar to context switch (but lighter weight)
  - Hardware saves some context on stack; Includes interrupted instruction if restart needed
  - Enters kernel at a specific point; kernel then figures out which *interrupt handler* should run
  - Execution resumes with special “iret” instruction
- Many different types of interrupts

Types of Interrupts

- Asynchronous
  - From external source, such as I/O device
  - Not related to instruction being executed
- Synchronous (also called *exceptions*)
  - *Processor-detected* exceptions:
    - *Faults* — correctable; offending instruction is *retried*
    - *Traps* — often for debugging; instruction is *not* retried
    - *Aborts* — major error (hardware failure)
  - *Programmed* exceptions:
    - Requests for kernel intervention (software intr/syscalls)

Faults

- Instruction would be illegal to execute
- Examples:
  - Writing to a memory segment marked ‘read-only’
  - Reading from an unavailable memory segment (on disk)
  - Executing a ‘privileged’ instruction
- Detected *before* incrementing the IP
- The causes of ‘faults’ can often be ‘fixed’
- If a ‘problem’ can be remedied, then the CPU can just resume its execution-cycle

Traps

- A CPU might have been programmed to automatically switch control to a ‘debugger’ program after it has executed an instruction
- That type of situation is known as a ‘trap’
- It is activated *after* incrementing the IP

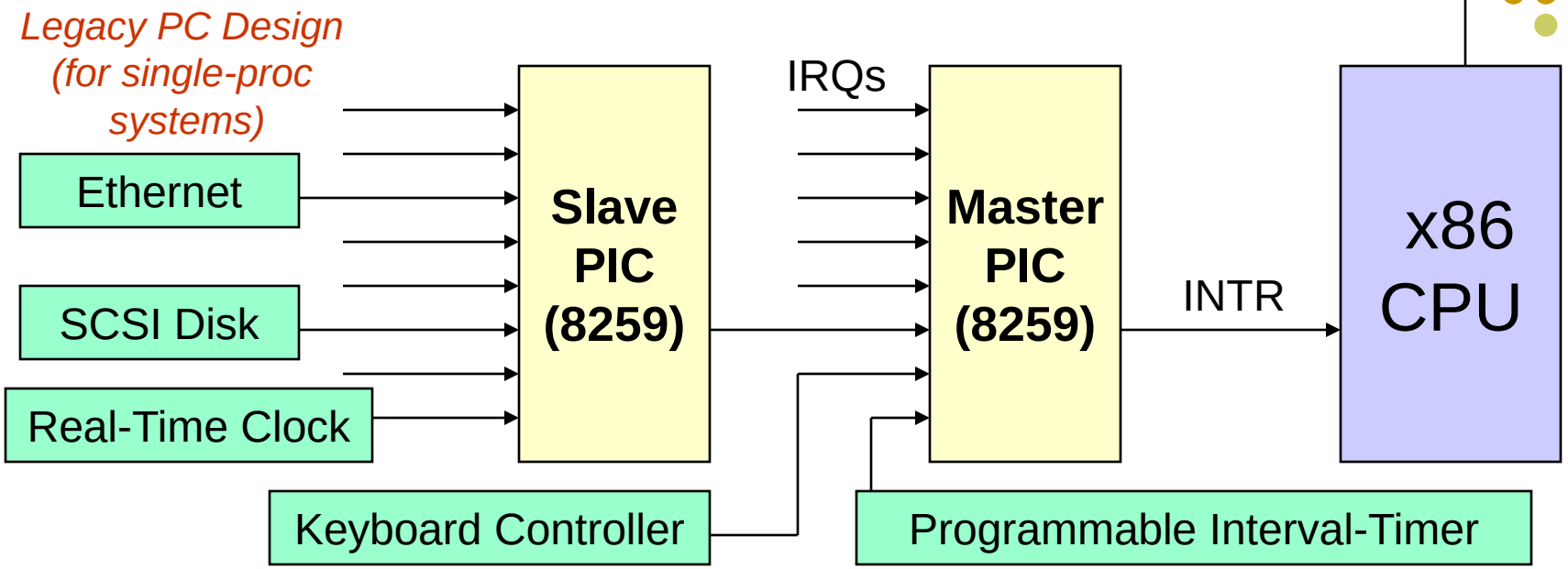
Error Exceptions

- Most error exceptions — divide by zero, invalid operation, illegal memory reference, etc. — translate directly into signals
- This isn't a coincidence. . .
- The kernel's job is fairly simple: send the appropriate signal to the current process
  - `force_sig(sig_number, current);`
- That will probably kill the process, but that's not the concern of the exception handler
- One important exception: page fault
- An exception can (infrequently) happen in the kernel
  - `die(); // kernel oops`

Intel-Reserved ID-Numbers

- Of the 256 possible interrupt ID numbers, Intel reserves the first 32 for ‘exceptions’
- OS’s such as Linux are free to use the remaining 224 available interrupt ID numbers for their own purposes (e.g., for service-requests from external devices, or for other purposes such as system-calls)
- Examples:
  - 0: divide-overflow fault
  - 6: Undefined Opcode
  - 7: Coprocessor Not Available
  - 11: Segment-Not-Present fault
  - 12: Stack fault
  - 13: General Protection Exception
  - 14: Page-Fault Exception

Interrupt Hardware



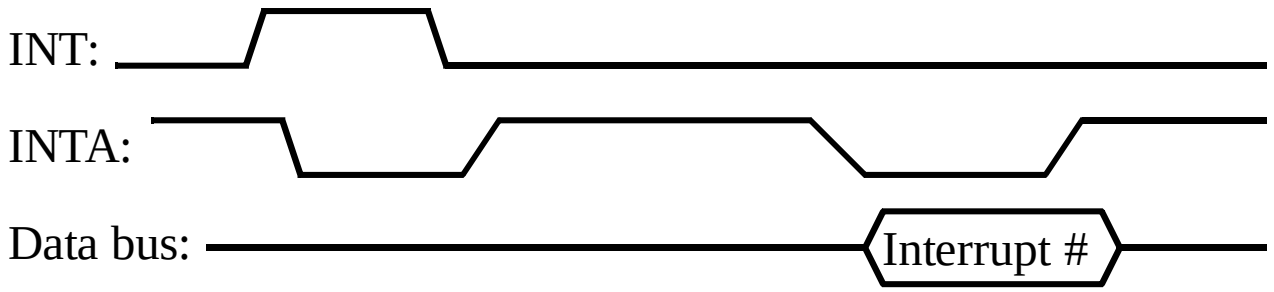
- I/O devices have (unique or shared) *Interrupt Request Lines* (IRQs)
- IRQs are mapped by special hardware to *interrupt vectors*, and passed to the CPU
- This hardware is called a *Programmable Interrupt Controller* (PIC)

The `Interrupt Controller'

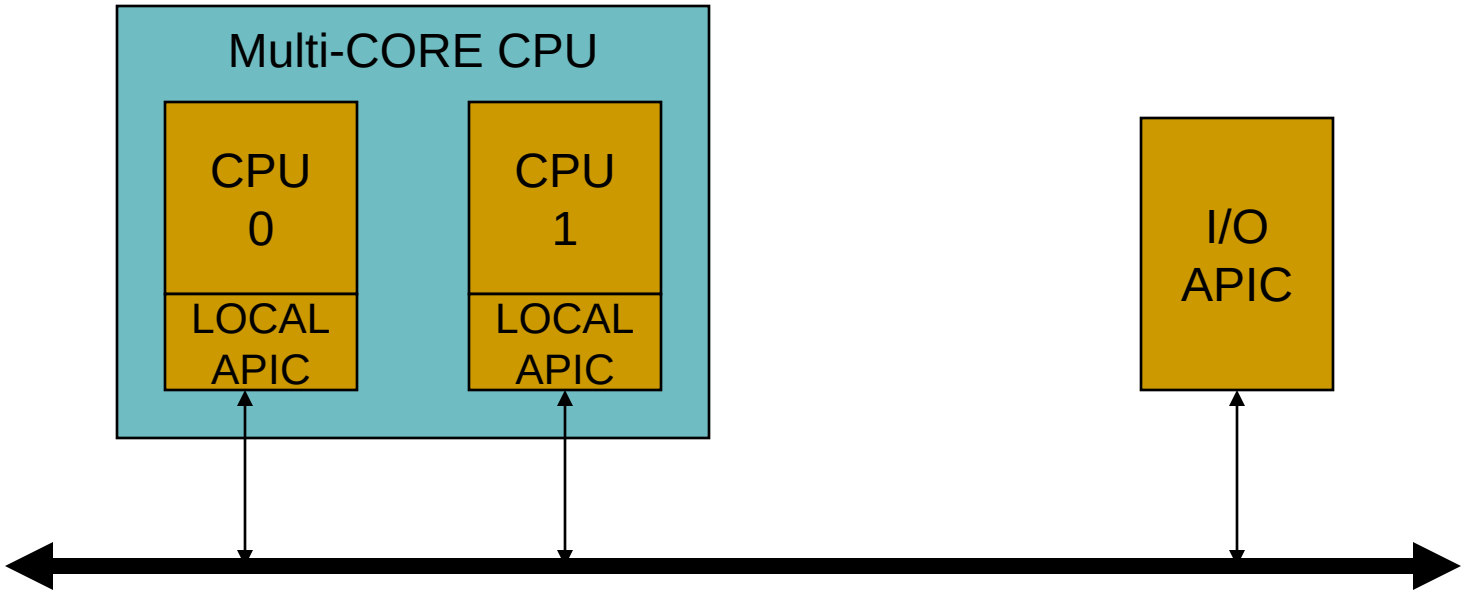
- Responsible for telling the CPU when a specific external device wishes to 'interrupt'
  - Needs to tell the CPU *which* one among several devices is the one needing service
- PIC translates IRQ to *vector*
  - Raises interrupt to CPU
  - Vector available in register
  - Waits for ack from CPU
- Interrupts can have varying priorities
  - PIC also needs to prioritize multiple requests
- Possible to "mask" (disable) interrupts at PIC or CPU
- Early systems cascaded two 8 input chips (8259A)

Example: Interrupts on 80386

- 80386 core has one interrupt line, one interrupt acknowledge line
- Interrupt sequence:
  - Interrupt controller raises INT line
  - 80386 core pulses INTA line low, allowing INT to go low
  - 80386 core pulses INTA line low again, signaling controller to put interrupt number on data bus



Multiple Logical Processors



Advanced Programmable Interrupt Controller is needed to perform ‘routing’ of I/O requests from peripherals to CPUs

(The legacy PICs are masked when the APICs are enabled)

Maximizing Parallelism

- You want to keep all I/O devices as busy as possible
- In general, an I/O interrupt represents the end of an operation; another request should be issued as soon as possible
- Most devices don’t interfere with each others’ data structures; there’s no reason to block out other devices

Handling Nested Interrupts

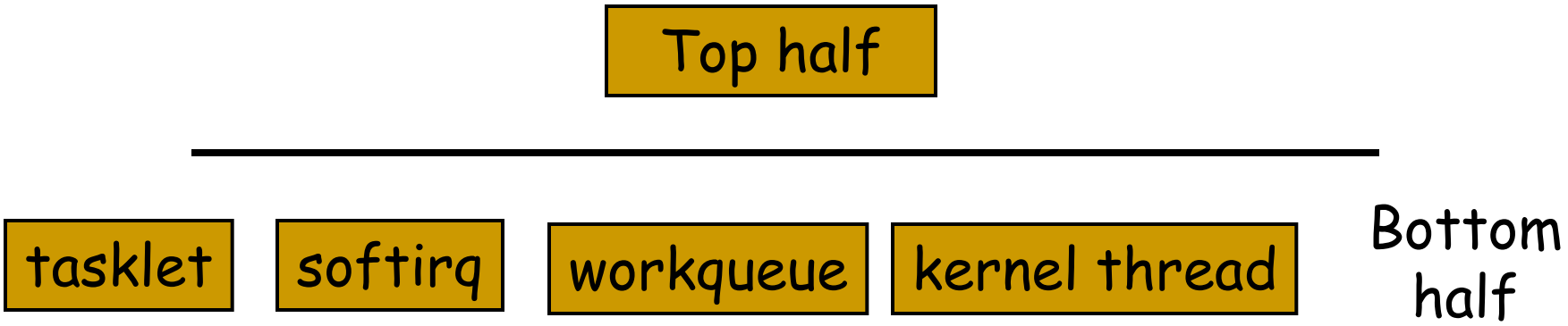
- As soon as possible, unmask the global interrupt
- As soon as reasonable, re-enable interrupts from that IRQ
- But that isn’t always a great idea, since it could cause re-entry to the same handler
- IRQ-specific mask is not enabled during interrupt-handling

Nested Execution

- Interrupts can be interrupted
  - By different interrupts; handlers need not be reentrant
  - No notion of priority in Linux
  - Small portions execute with interrupts disabled
  - Interrupts remain pending until acked by CPU
- Exceptions can be interrupted
  - By interrupts (devices needing service)
- Exceptions can nest two levels deep
  - Exceptions indicate coding error
  - Exception code (kernel code) shouldn’t have bugs
  - Page fault is possible (trying to touch user data)

Interrupt Handling Philosophy

- Do as little as possible in the interrupt handler
- Defer non-critical actions till later
- Structure: top and bottom halves
  - Top-half: do minimum work and return (ISR)
  - Bottom-half: deferred processing (softirqs, tasklets, workqueues, kernel threads)



Interrupt Stack

- When an interrupt occurs, what stack is used?
  - Exceptions: The *kernel stack* of the current process, whatever it is, is used (There's always some process running — the "idle" process, if nothing else)
  - Interrupts: hard IRQ stack (1 per processor)
  - SoftIRQs: soft IRQ stack (1 per processor)
- These stacks are configured in the IDT and TSS at boot time by the kernel

Hardware Handling

- On entry:
  - Which vector?
  - Get corresponding descriptor in IDT
  - Find specified descriptor in GDT (for handler)
  - Check privilege levels (CPL, DPL)
    - If entering kernel mode, set kernel stack
  - Save eflags, cs, (original) eip on stack
- Jump to appropriate handler
  - Assembly code prepares C stack, calls handler
- On return (i.e. iret):
  - Restore registers from stack
  - If returning to user mode, restore user stack
  - Clear segment registers (if privileged selectors)

Interrupt Handling

- More complex than exceptions
  - Requires registry, deferred processing, etc.
- Three types of actions:
  - Critical: Top-half (interrupts disabled – briefly!)
    - Example: acknowledge interrupt
  - Non-critical: Top-half (interrupts enabled)
    - Example: read key scan code, add to buffer
  - Non-critical deferrable: Do it “later” (interrupts enabled)
    - Example: copy keyboard buffer to terminal handler process
    - Softirqs, tasklets