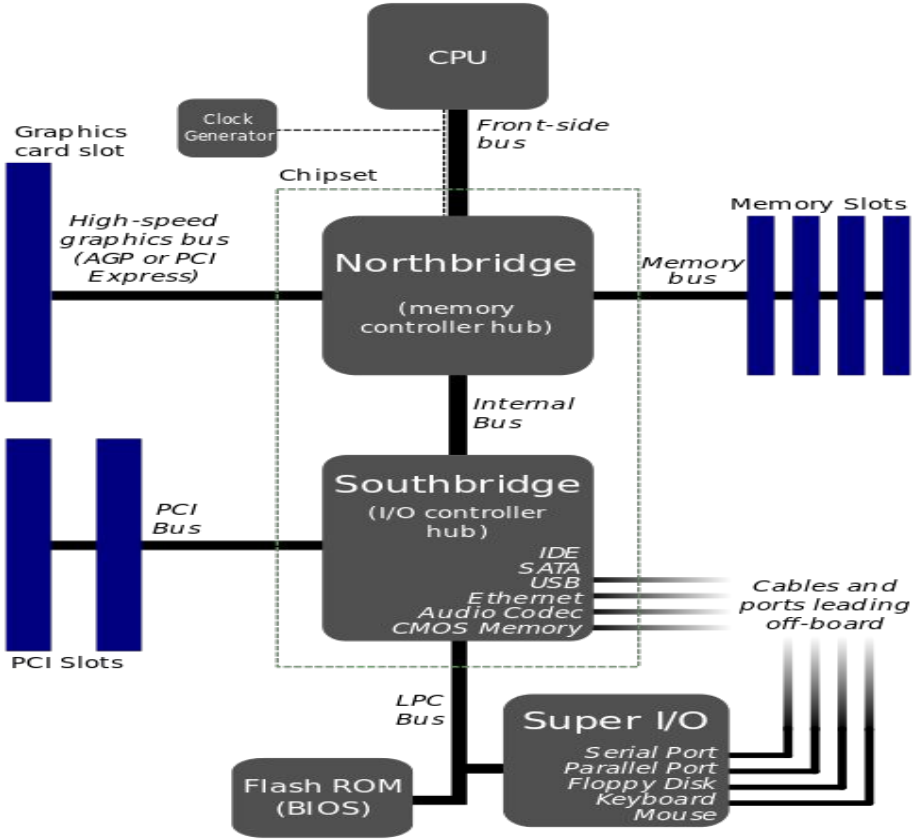
I/O and Device Drivers

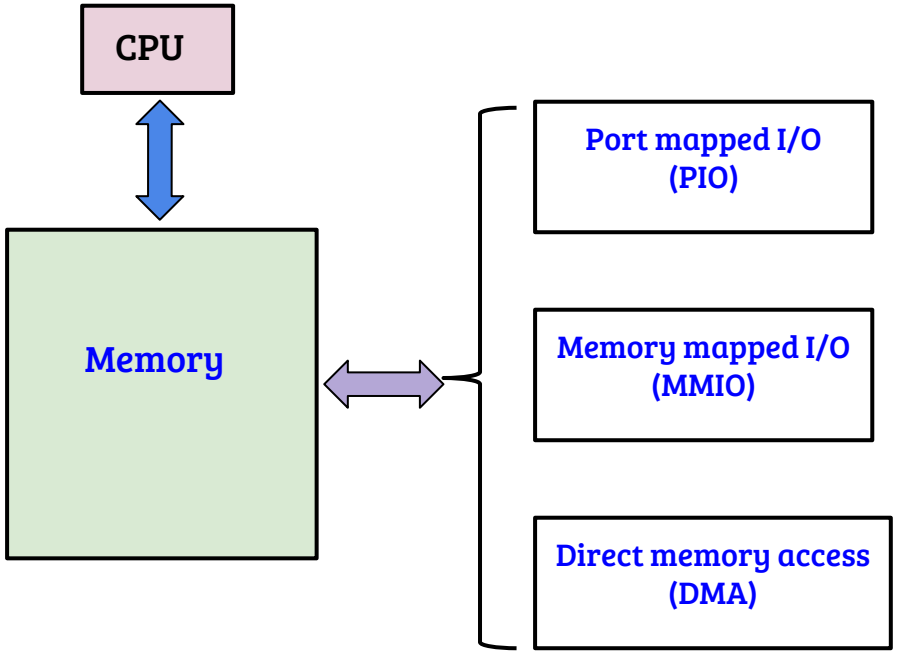
I/O architecture



- ➔ CPU access to I/O devices and memory using bus interconnects
- ➔ Why I/O devices can not be operated like memory?
- ➔ What are the different ways to access I/O devices?

Image source: https://commons.wikimedia.org/wiki/

I/O device interfacing



- ➔ Why so many interfaces?
- ➔ I/O port address
  - ◆ Device registers/pins mapped to port address
  - ◆ != memory address
  - ◆ Limited instructions (in, out)
- ➔ Memory mapped address
  - ◆ Device registers mapped to memory
  - ◆ All instructions available
- ➔ DMA: bulk-data transfer w/o CPU involvement

I/O access: Programmer's perspective

- ➔ Device clock and CPU clock mismatch

- ➔ CPU and compiler optimizations
  - ◆ Instruction reordering
  - ◆ Data caching (in registers)
  - ◆ Examples
- ➔ Some familiar keywords
  - ◆ Volatile
  - ◆ Memory barriers
- ➔ User-mode access: iopl and ioperm

How to find my device?

- BIOS fixed ports, memory maps, IRQ lines
  - Inflexible but unavoidable
  - Why?
- Alternate organization (idea of PCI)
  - A tree organization with fixed root
  - Can be explored from the root
  - Expandable architecture
- lscpi
  - Domain, bus, device, function

PCI devices

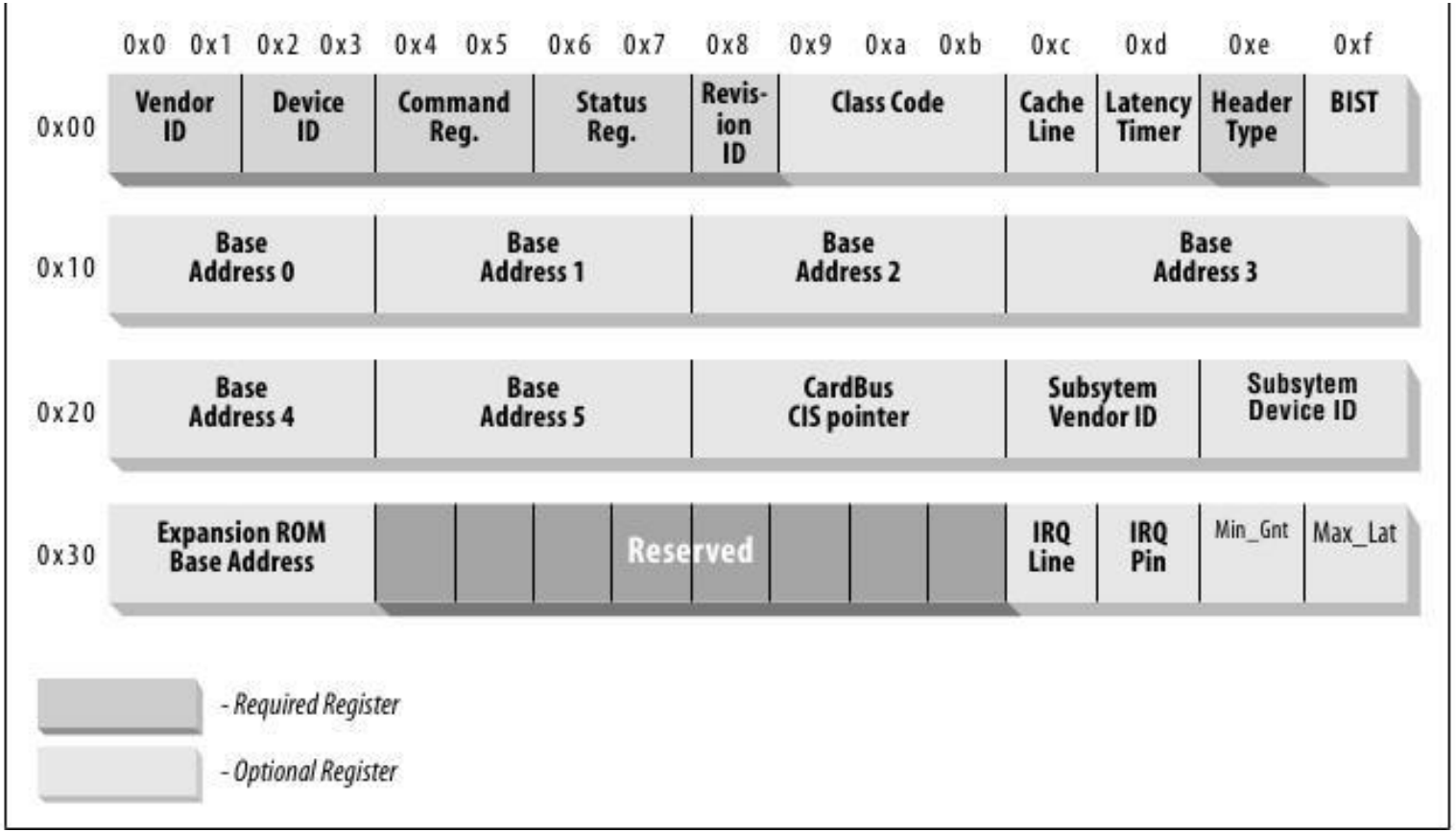


Image source: Linux Device Drivers, Ch12

Writing PCI device drivers: probe

- Register PCI device driver with {deviceID, vendorID, probefn, removefn …}
- PCI core calls the probe function during device listing (scan)
  - Boot time, hot-plug or forced rescan

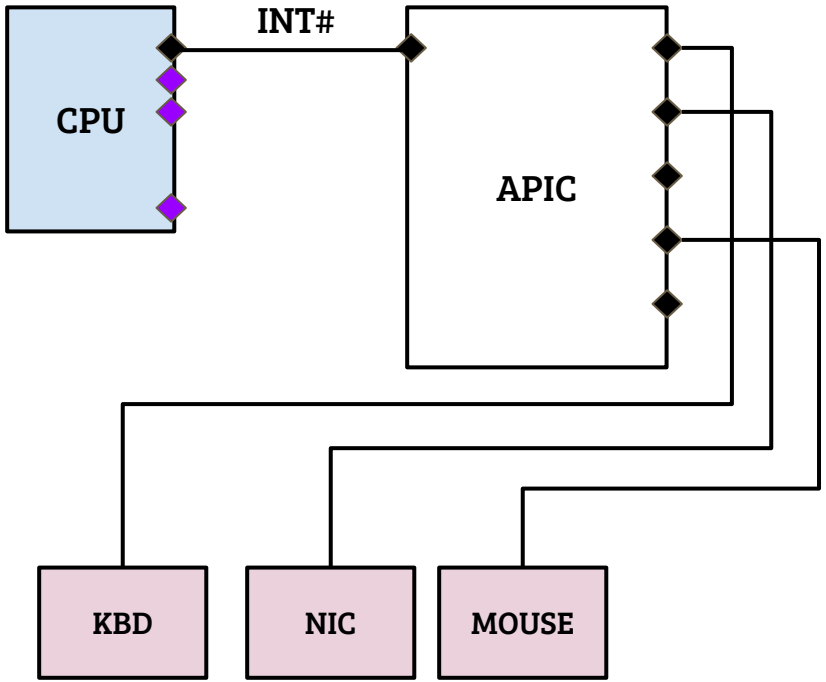
- The probe( ) method
  - Access PCI configuration space
  - Access the MMIO/PIO information
  - Setup device specific state
  - Associate IRQ with interrupt handler
  - Interface with upper layer
- Wait, either invoked from upper layer or device interrupt

Interrupt handling

- ➔ Why interrupts?
- ➔ CPU can be interrupted, but has limited number of pins
- ➔ Solution?

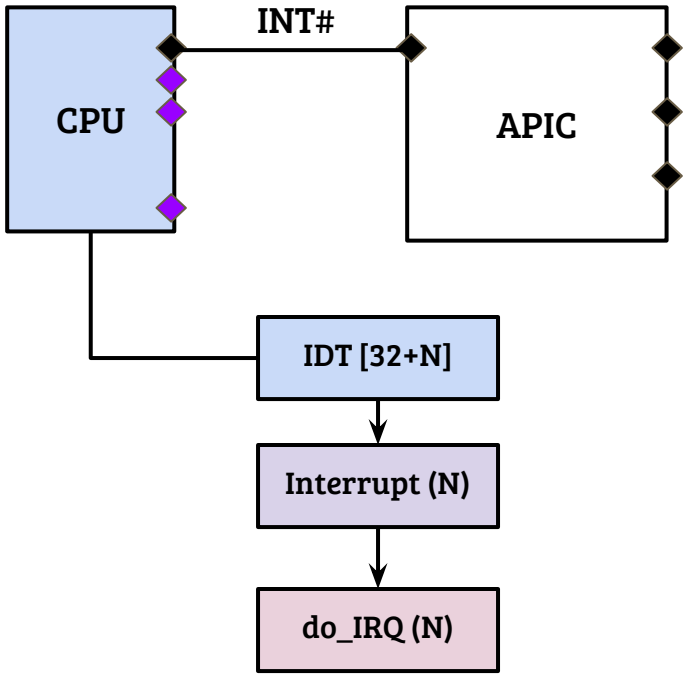
- ➔ Interrupt handling challenges and tradeoffs
  - ◆ Non-deterministic event arrival
  - ◆ Responsiveness
  - ◆ CPU overheads
  - ◆ Scalability
  - ◆ … and the application progress

Interrupt architecture - PIC and APIC



- ➔ APIC raise CPU interrupt line on receipt of device interrupt
- ➔ IRQ line to ISR?
- ➔ Waits for acknowledgement before clearing the line
- ➔ Selective disabling possible
  - != cli (CPU interrupt disable)
  - New interrupts not lost
- ➔ CPU should acknowledge the interrupt quickly!
- ➔ Multiprocessor systems?

Interrupt handling



- ➔ IDT configured to load the interrupt execution context
- ➔ Interrupt (): assembly code at entry_64.S
- ➔ do_IRQ( ) calls the real device interrupt handler
- ➔ Device driver should acknowledge the interrupt quickly!
- ➔ Not all interrupts can be handled quickly, e.g., NIC RCV

Interrupt handling: SoftIRQ

- ➔ Carry out deferrable operations
- ➔ Just like an interrupt, can be raised, disabled, enabled, masked
- ➔ Actually executed by the local CPU kernel thread
  - ◆ Ksoftirqd, one per CPU
  - ◆ A loop checking for pending softIRQ
  - ◆ Often scheduled on irq_exit( )
- ➔ Example: Protocol processing of a network packet
  - ◆ Update ring buffer s/w pointers
  - ◆ Interrupt handler (RCV) raises NET_RX_SOFTIRQ
  - ◆ On ISR completion, the local ksoftirqd thread is scheduled
  - ◆ Calls the handler (do_softirq)

Interrupt on multiprocessor systems: IOAPIC