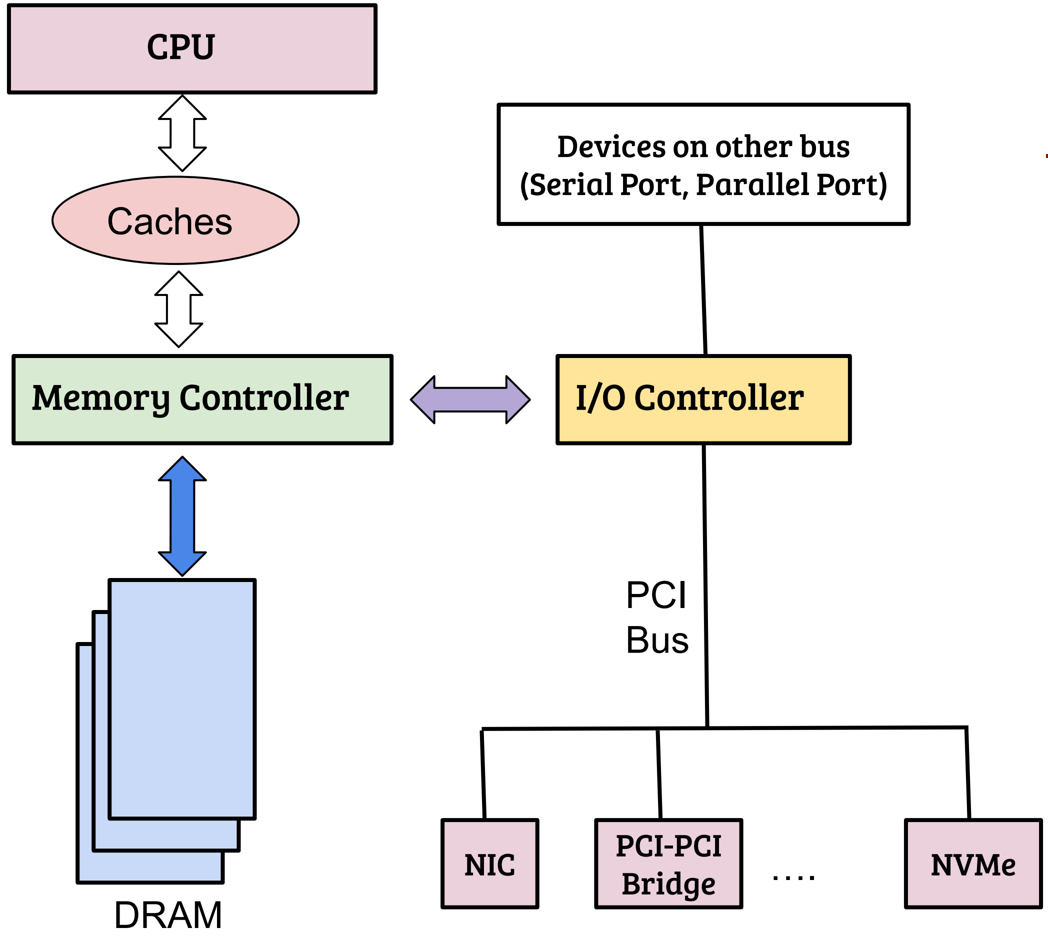
CS614: Linux Kernel Programming

I/O and Device drivers

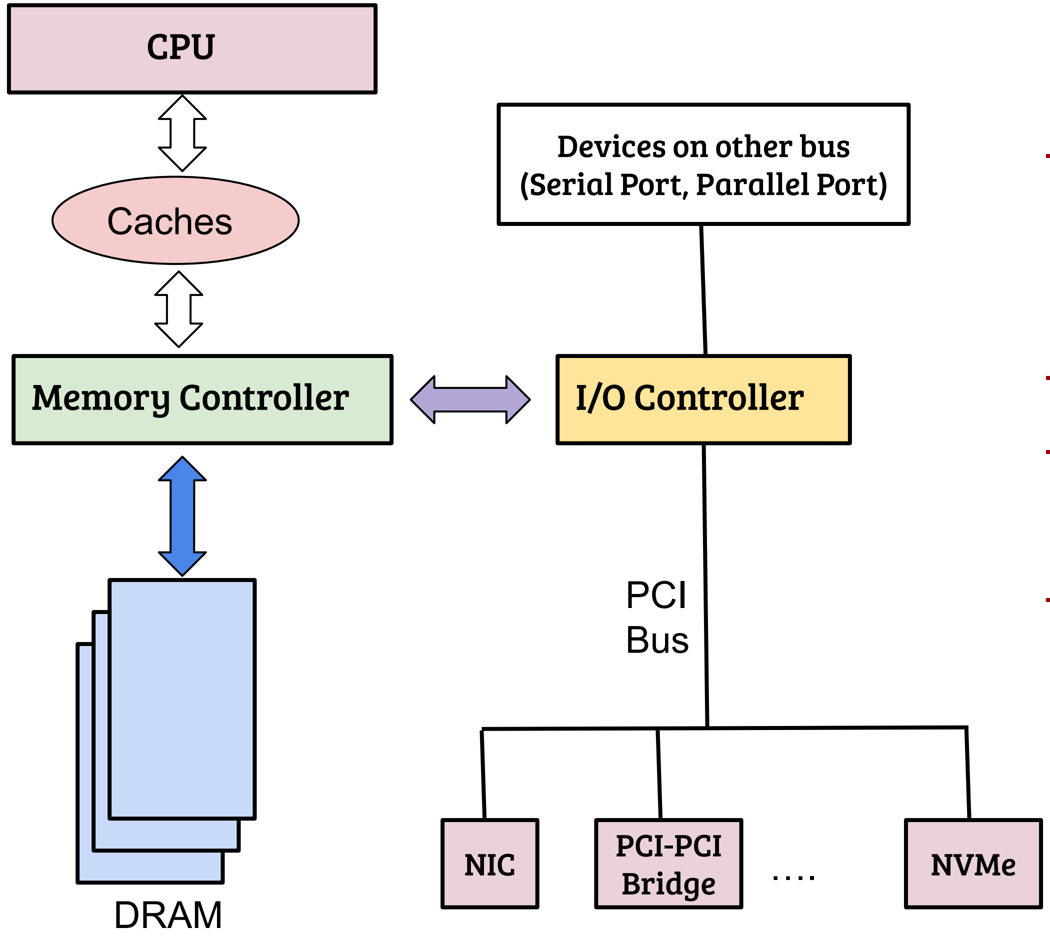
Debadatta Mishra, CSE, IIT Kanpur

I/O device interfacing (example organization)



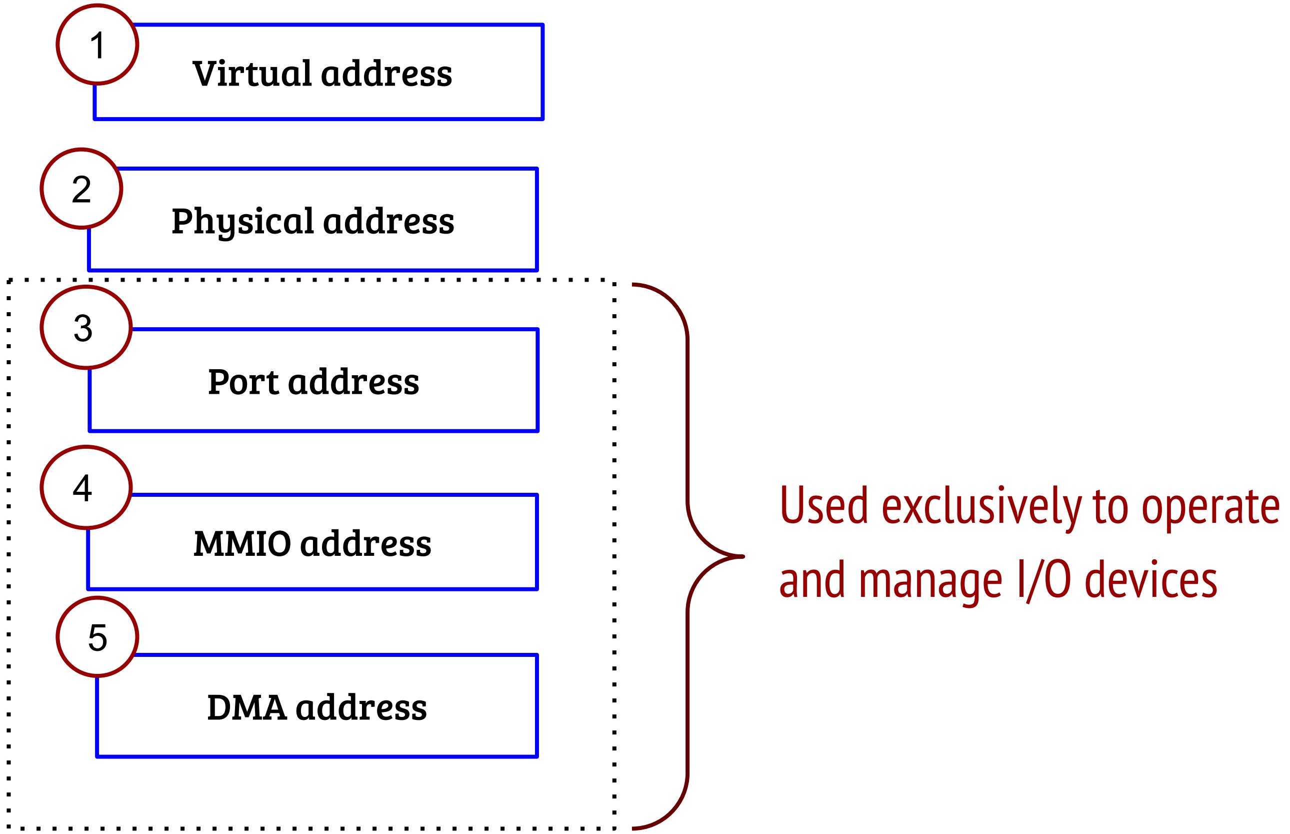
- Typically, I/O devices consists of
  - Registers (data regs (r/w), command regs, status regs etc.)
  - Memory (in-device memory, e.g., GPU memory)
  - Logic and processing (e.g, calculate a packet checksum)

I/O device interfacing (example organization)

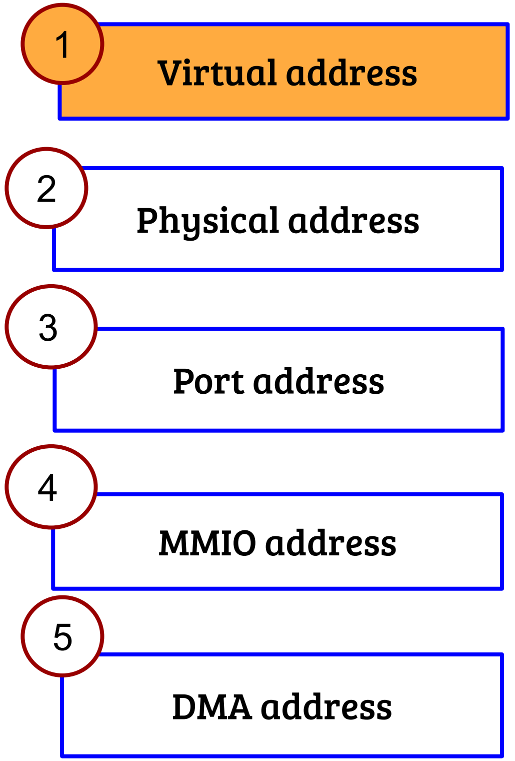


- To configure and use I/O devices, CPU should be able to operate the I/O devices (Device regs and memory)
- How to address different I/O devices?
- How to address different device resources (regs and memory)?
- Can we address the I/O devices using memory load/store instructions?

Address types in kernel

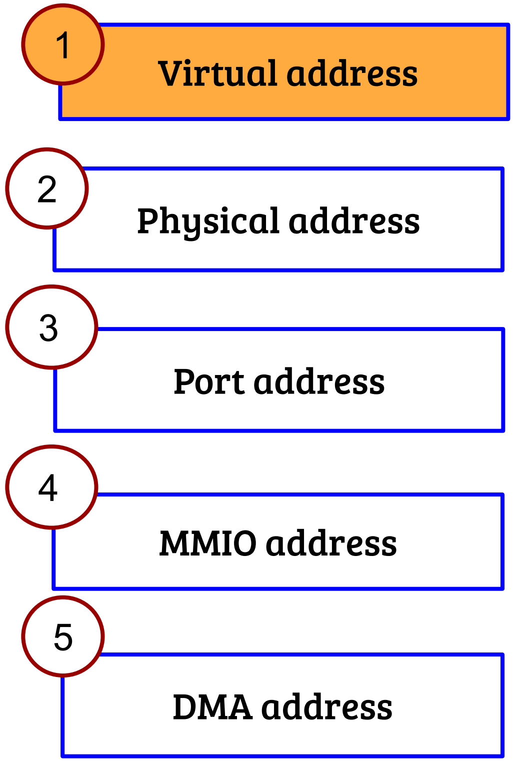


Kernel virtual address



- Direct mapping of physical memory (64TB)
  - Conversion from virtual to physical and vice-a-versa can be done using macros like __va(paddr) and __pa(vaddr)

Kernel virtual address



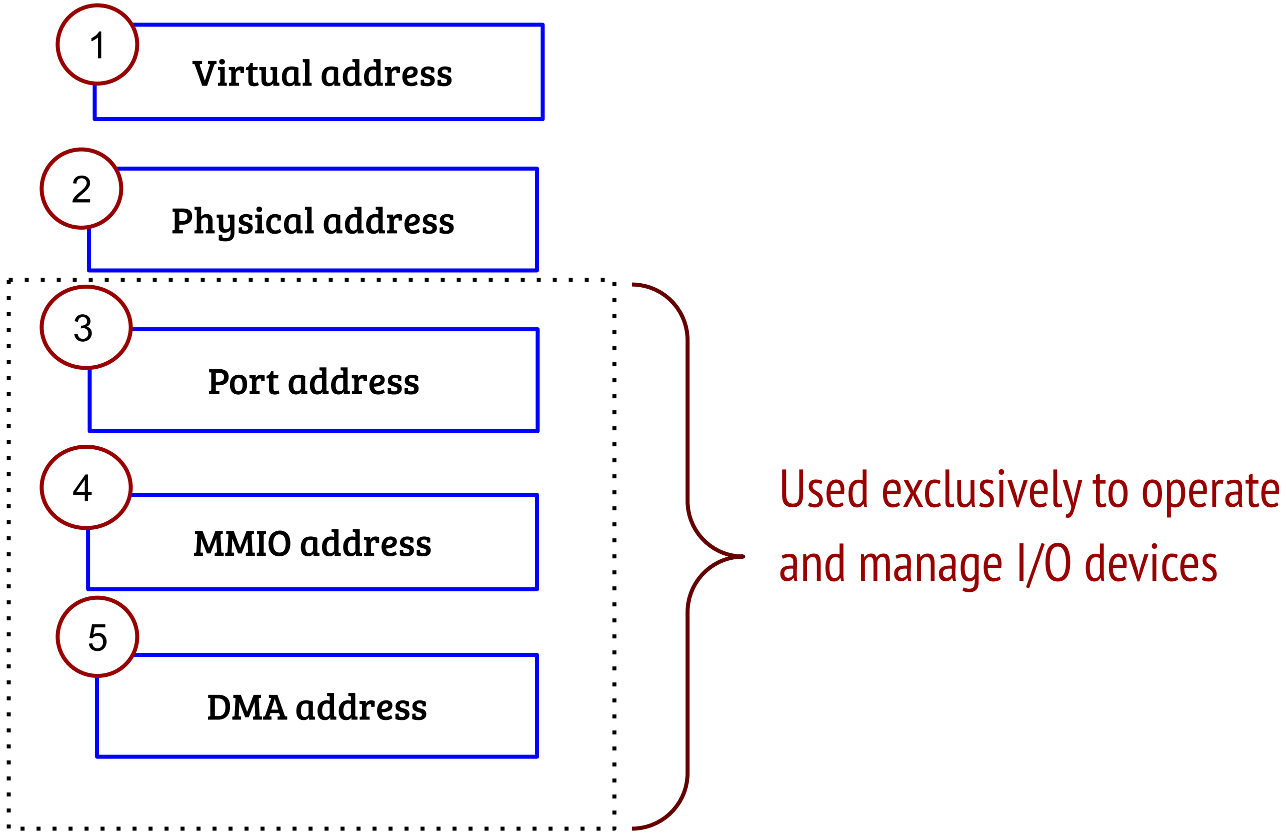
- Direct mapping of physical memory (64TB)
  - Conversion from virtual to physical and vice-a-versa can be done using macros like __va(paddr) and __pa(vaddr)
- Physically discontinuous virtual address
  - Allocated used vmalloc( )
  - Useful when you allocate large contiguous kernel virtual address
  - Legacy: 32-bit systems required temporary virtual addresses a lot (check out highmem)

Physical address in kernel

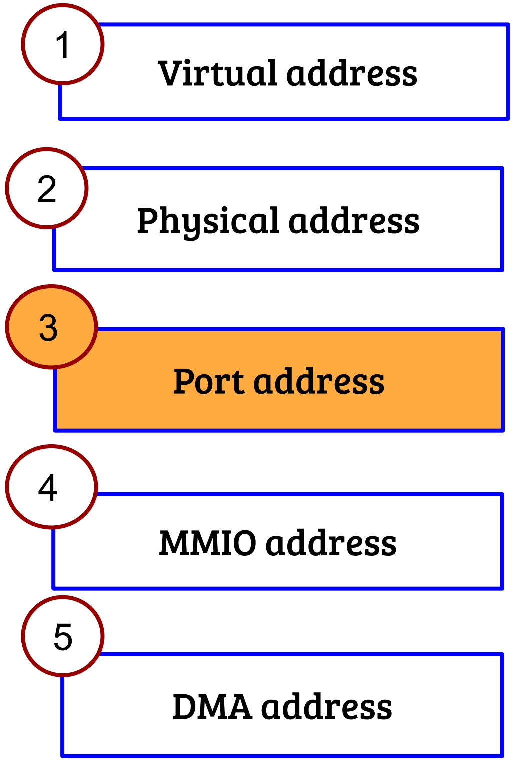
1. Virtual address
2. Physical address
3. Port address
4. MMIO address
5. DMA address

- Two commonly used (almost interchangeable) terms
  - Page: A *struct page* type
  - Physical Frame Number (PFN): *unsigned long*
  - APIs: pfn_to_page, page_to_pfn etc.
  - How does the conversion happen?
- At the lowest level, physical allocation done through page allocation APIs (alloc_page, free_page etc.)
- Page structure contains information like mapcount, usage count etc.

Address types in kernel



Port addressing



- Device registers mapped by BIOS to port addresses
- Port addresses can be accessed directly without using page table mapping
- However, port addresses are
    - Not memory addresses
    - Only I/O instructions (in, out) are allowed
- $cat /proc/ioports
- OSes have to use some hard coded port addresses (created by BIOS mapping), it is unavoidable!
- Example: Serial console

Port I/O access

- Instructions: inb, outb, inw, outw, inl, outl
- Example: “outb $0x3F8, $0x5” → Write five to the port address 0x3F8

Port I/O access

- Instructions: inb, outb, inw, outw, inl, outl
- Example: "outb $0x3F8, $0x5" → Write five to the port address 0x3F8
- Important: Completion of a write instruction may not imply the intended I/O operation is completed (CPU and I/O speeds may not match!)
- Example: To print a string into a serial console using pio, writing back to back chars may result in data loss as the device may not handle the output at CPU speed
- How should the OS ensure completion of I/O actions?

Port I/O access

- Instructions: inb, outb, inw, outw, inl, outl
- Example: “outb $0x3F8, $0x5” → Write five to the port address 0x3F8
- Important: Completion of a write instruction may not imply the intended I/O operation is completed (CPU and I/O speeds may not match!)
- Example: To print a string into a serial console using pio, writing back to back chars may result in data loss as the device may not handle the output at CPU speed
- How should the OS ensure completion of I/O actions?

Port I/O access

- Instructions: inb, outb, inw, outw, inl, outl
- Example: “outb $0x3F8, $0x5” $\rightarrow$ Write five to the port address 0x3F8
- Important: Completion of a write instruction may not imply the intended I/O operation is completed (CPU and I/O speeds may not match!)
- Example: To print a string into a serial console using pio, writing back to back chars may result in data loss as the device may not handle the output at CPU speed
- How should the OS ensure completion of I/O actions?
    - If the device provides a “status” port, OS can check
    - What if the device manual suggest a particular speed for an operation?

Port I/O access

- Instructions: inb, outb, inw, outw, inl, outl
- Example: "outb \$0x3F8, \$0x5" ⟶ Write five to the port address 0x3F8
- Important: Completion of a write instruction may not imply the intended I/O operation is completed (CPU and I/O speeds may not match!)
- Example: To print a string into a serial console using pio, writing back to back chars may result in data loss as the device may not handle the output at CPU speed
- How should the OS ensure completion of I/O actions?
  - If the device provides a "status" port, OS can check
  - What if the device manual suggest a particular speed for an operation? Calibrate device clock speed and wait for device cycles mentioned in the specifications

Port I/O access

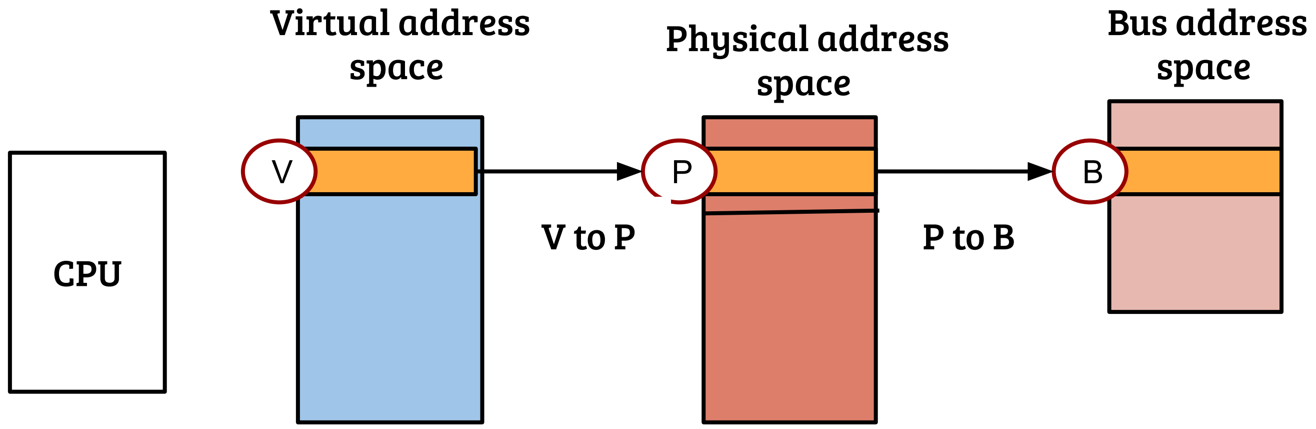
- Instructions: inb, outb, inw, outw, inl, outl
- Example: "outb $0x3F8, $0x5" → Write five to the port address 0x3F8
- Important: Completion of a write instruction may not imply the intended I/O operation is completed (CPU and I/O speeds may not match!)
- Example: To print a string into a serial console using pio, writing back to back chars may result in data loss as the device may not handle the output at CPU speed
- How should the OS ensure completion of I/O actions?
    - If the device provides a "status" port, OS can check
    - What if the device manual suggest a particular speed for an operation? Calibrate device clock speed and wait for device cycles mentioned in the specifications
- Driver programmer should be careful about reorderings! Use of "volatile" keyword and "fence" instructions in X86

Memory mapped I/O

1. Virtual address
2. Physical address
3. Port address
4. **MMIO address**
5. DMA address

- I/O registers/memory mapped into physical address space, can be accessed like memory
- What address to use, virtual or physical?
- What extra care to be taken while accessing MMIO addresses?

Memory mapped I/O



- During device discovery, kernel maintains a device to MMIO space (/proc/iomem)
- Device driver must map the PA to V before access
- Kernel source: ioremap( ), ioread32( )
- Example: gemOS APIC setup

Memory mapped I/O

1. Virtual address
2. Physical address
3. Port address
4. **MMIO address**
5. DMA address

- I/O registers/memory mapped into physical address space, can be accessed like memory
- What address to use, virtual or physical?
- Virtual address
- What extra care to be taken while accessing MMIO addresses?
- Correctly timing the accesses, compiler optimizations, OOO processing

PIO and MMIO: User mode vs. Kernel mode

- Isolation requirements require I/O access restrictions from the user space
- However, in some cases, it may be required; Can the OS allow I/O access from user mode?
- Port I/O?
- MMIO?

PIO and MMIO: User mode vs. Kernel mode

- Isolation requirements require I/O access restrictions from the user space
- However, in some cases, it may be required; Can the OS allow I/O access from user mode?
- Port I/O?
  - In intel X86 systems, IOPL bit in the flags register can be used to control access
  - For finer granularity control, I/O permission bitmap can be configured
- MMIO?

PIO and MMIO: User mode vs. Kernel mode

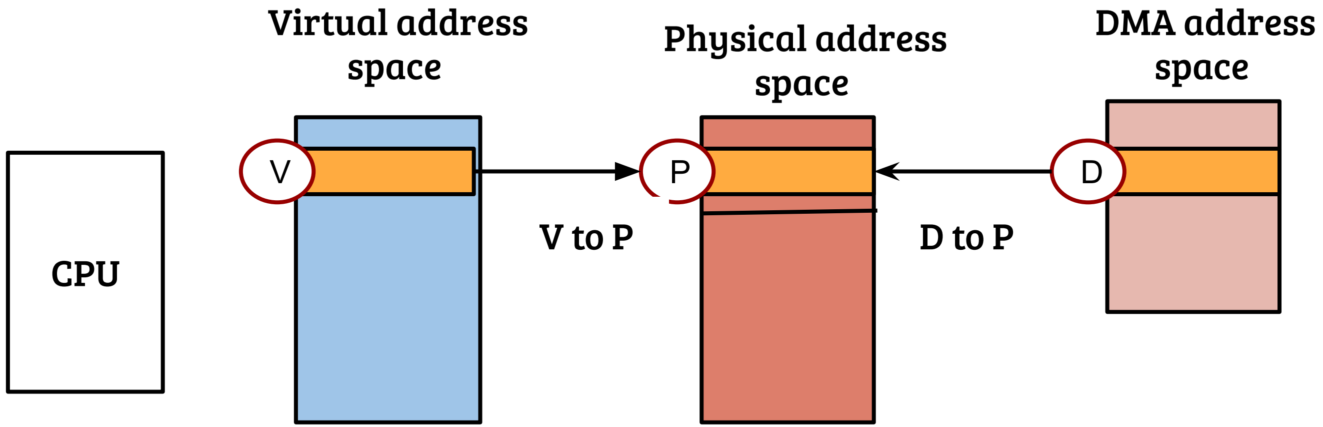
- Isolation requirements require I/O access restrictions from the user space
- However, in some cases, it may be required; Can the OS allow I/O access from user mode?
- Port I/O?
  - In intel X86 systems, IOPL bit in the flags register can be used to control access
  - For finer granularity control, I/O permission bitmap can be configured
- MMIO?
  - Restriction to MMIO is based on page level protections
  - If the OS maps a MMIO address to user virtual address, it can be accessed from the user mode
  - Challenge: MMIO address for different devices may belong to the same page

Direct memory access (DMA)

1. Virtual address
2. Physical address
3. Port address
4. MMIO address
5. **DMA address**

- DMA can be used if
  - DMA controller is available
  - Device supports DMA
- DMA addresses are generated and used by DMA controller
- Can be different from physical address if IOMMU is used

DMA contd.

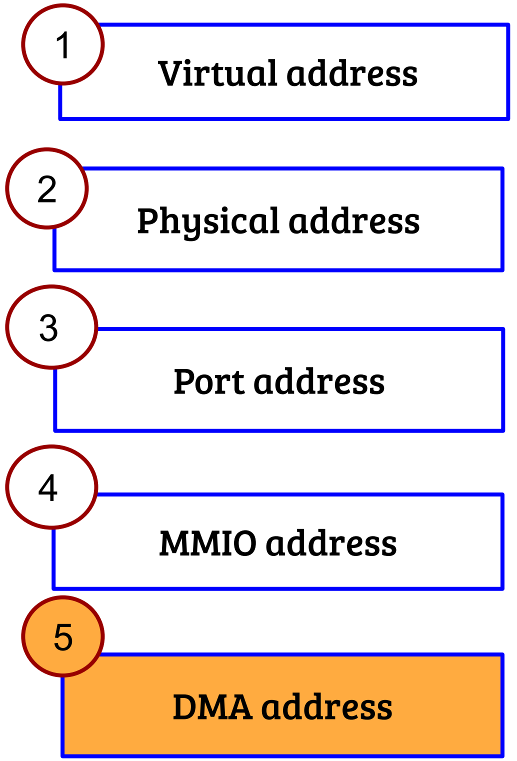


- Device driver allocates a buffer (VA = V, PA = P), no lazy allocation allowed!
- In non-IOMMU systems, device can use P directly
- With IOMMU, mapping must be setup between D $\longrightarrow$ P using API's like *dma_map_single*
- Why device driver programmer has to worry about the DMA address?

DMA and interrupt handling example

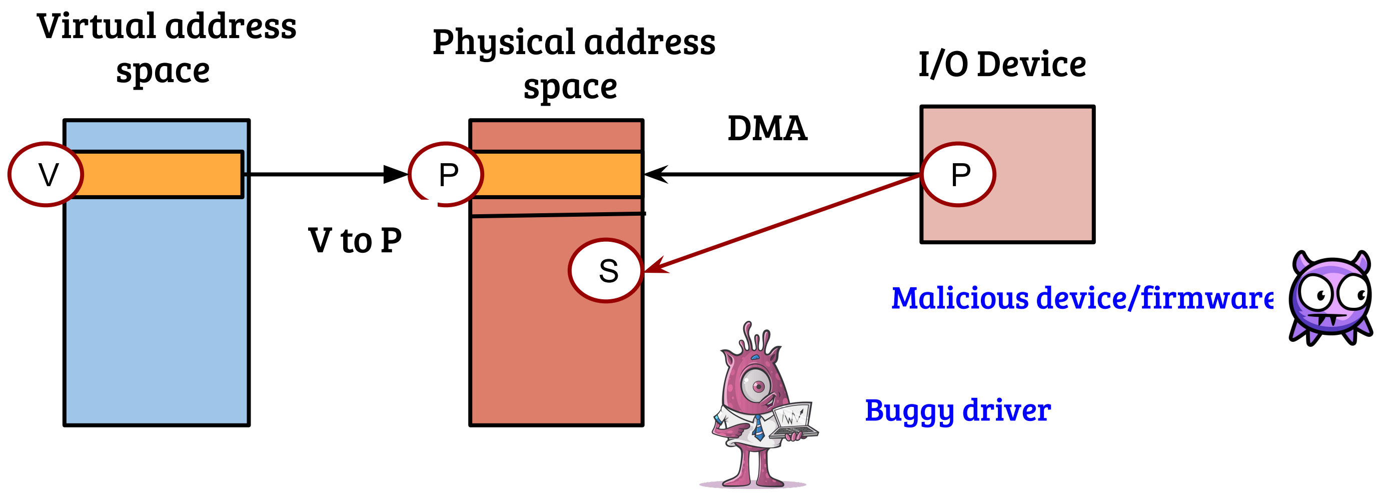
```
setup_one_rcv(NIC *nic){
    dma_addr_t *mapping;
    mapping = dma_map_single(nic->dev, nic->buff_va, nic-> len, DMA_FROM_DEVICE);
    nic->rcv_dma = mapping;
    mmio_nic(nic, DEVICE_SET_DMA);
}

irq_rcv_one(NIC *nic){
    dma_unmap_single(nic->dev, nic->buff_va, nic-> len, DMA_FROM_DEVICE);
    do_tcp_ip(nic->buff_va, nic->len);
 }
```

Direct memory access (DMA)



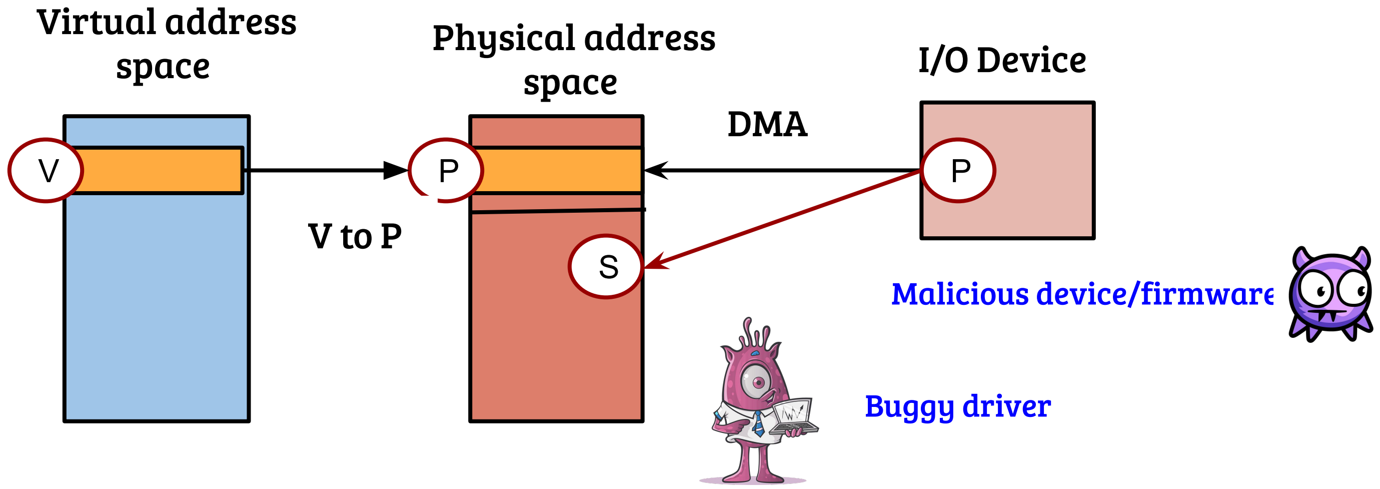
- Virtual addresses used by DMA should be mapped (don’t use vmalloc( ) address)
- DMA mapping can be of two types
  - Consistent/Coherent: mostly used throughout the driver lifetime
  - Streaming/inconsistent: used to configure receive buffer of a NIC
- Refer to kernel documentation (Documentation/core-api/dma-api-howto.txt) for details

Security issue with DMA



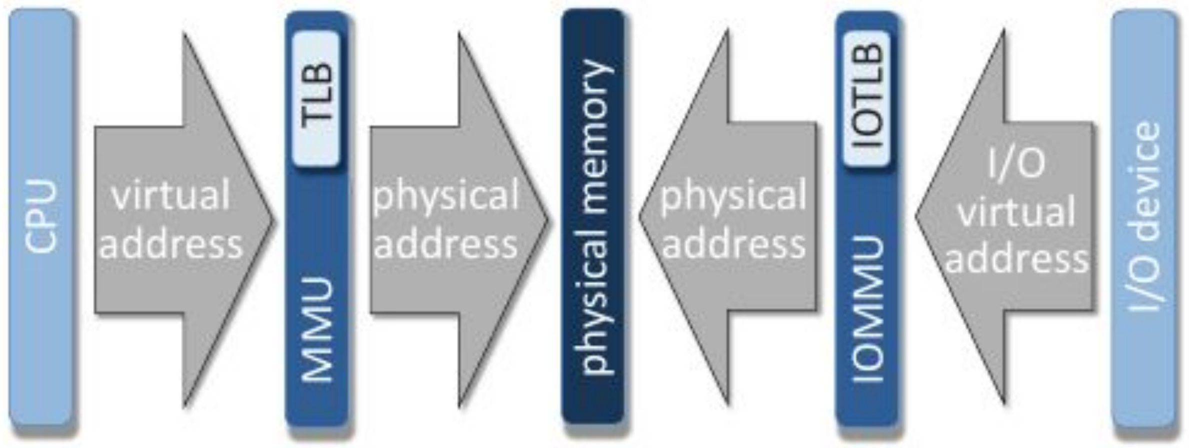
- I/O devices can access arbitrary memory locations
- Compromised security, information disclosure
- How to address this issue?

Security issue with DMA



- I/O devices can access arbitrary memory locations
- Compromised security, information disclosure
- How to address this issue? A layer of translation for I/O devices a.k.a. IOMMU

Introduction of I/O virtual address (IOVA) 1

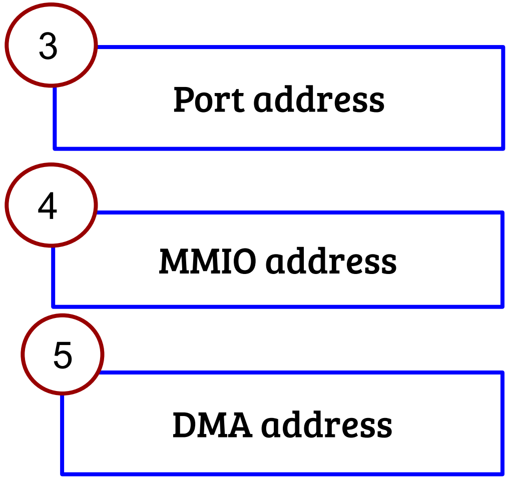


[1]

- In a nutshell, I/O devices are treated like a user process
- The OS associates the physical address with an IOVA and setup the IOVA-to-PA mapping in IOMMU tables
- IOMMU table is similar to page tables (with a TLB!)

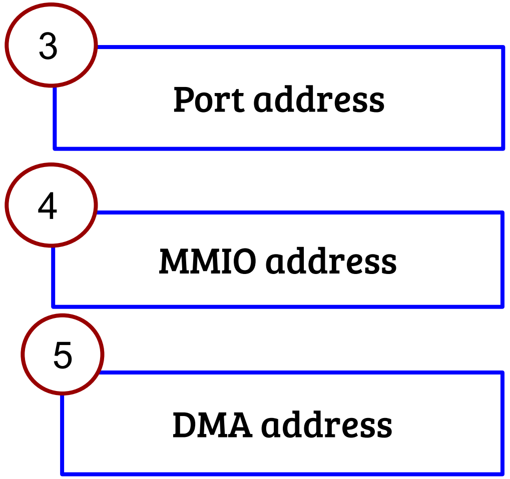
1. Malka et al. rIOMMU:Efficient IOMMU for I/O Devices that Employ Ring Buffers https://dl.acm.org/citation.cfm?id=2694355

Flexibility in I/O Addressing



- What kind of addressing provides more flexibility to the OS, considering address as a resource?

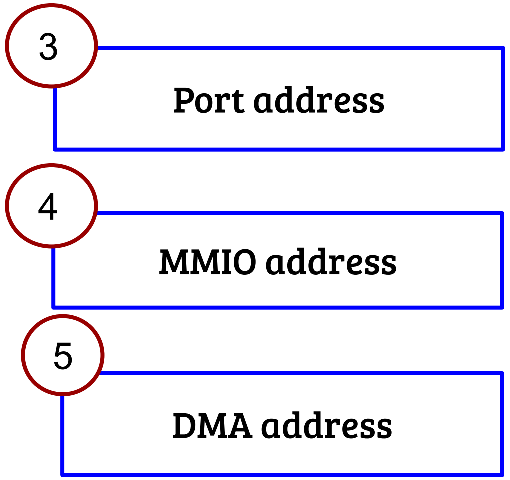
Flexibility in I/O Addressing



- What kind of addressing provides more flexibility to the OS, considering address as a resource?

  DMA allows maximum flexibility to the OS
- Can a device be initialized and operated only with DMA addressing?

Flexibility in I/O Addressing



- What kind of addressing provides more flexibility to the OS, considering address as a resource? DMA allows maximum flexibility to the OS
- Can a device be initialized and operated only with DMA addressing? No, because the DMA setup requires MMIO/PIO access
- How can the OS manage PIO and MMIO addresses in a flexible manner?

PCI Subsystem

- PCI can be viewed as tree-like organization of I/O devices
- Each device mapped to PCI bus can be examined based on the IDs (device, vendor etc.)

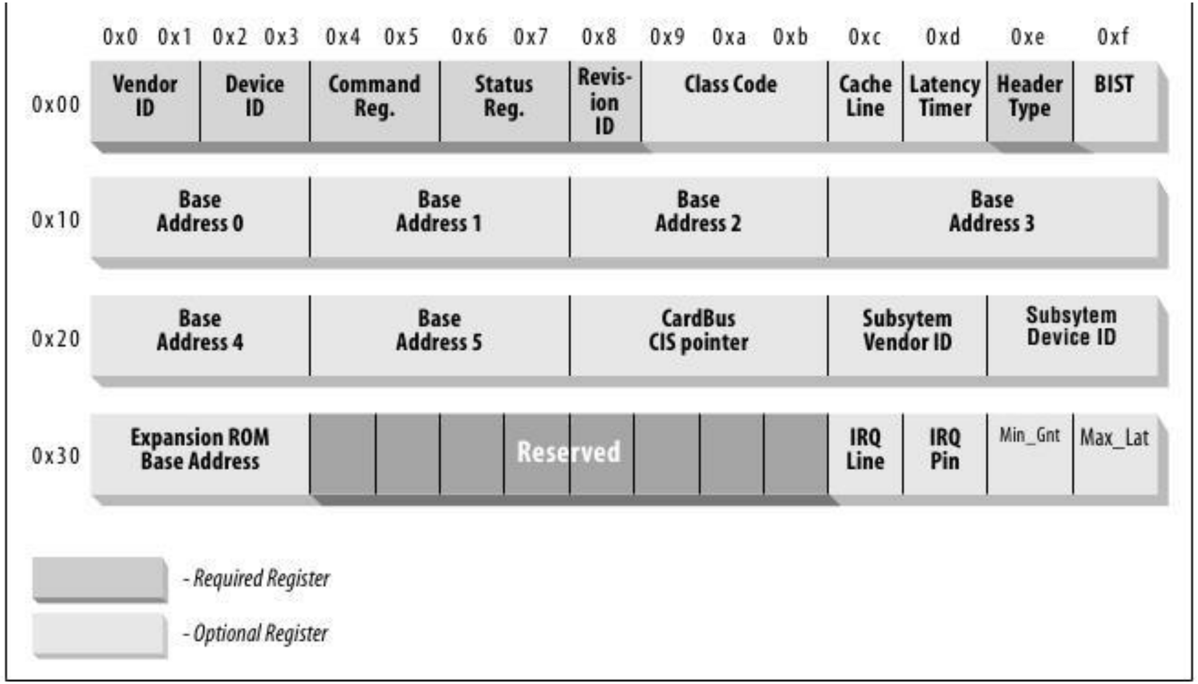


Image source: Linux Device Drivers, Ch12

PCI Layout

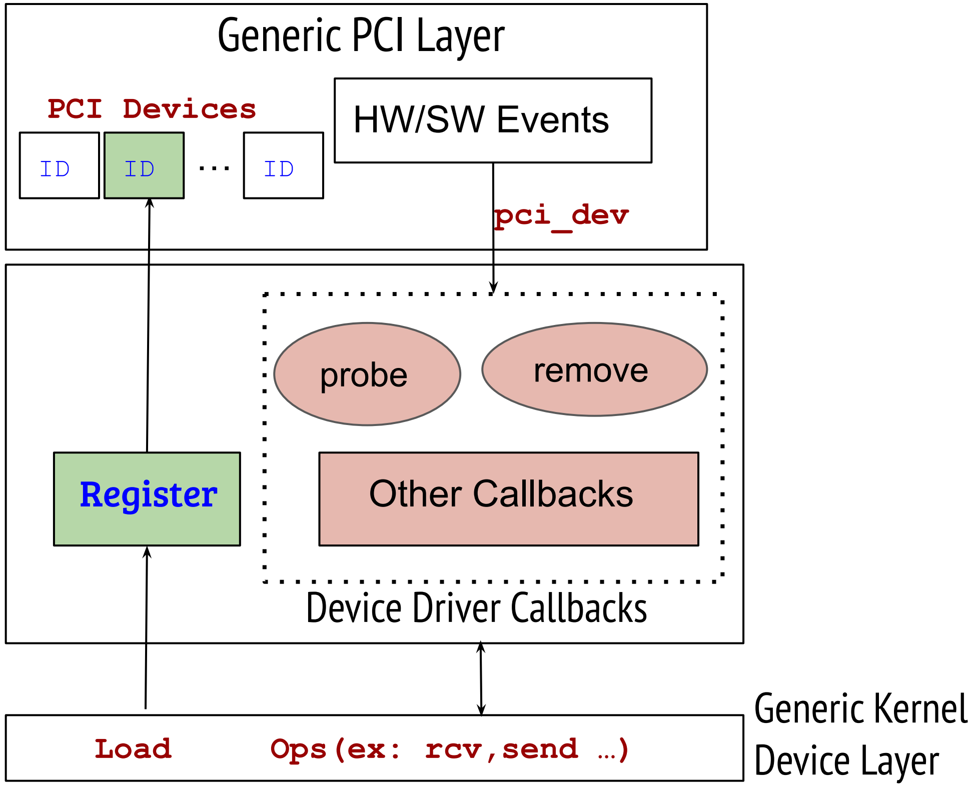
- PCI can be viewed as tree-like organization of I/O devices
- Each device mapped to PCI bus can be examined based on the IDs (device, vendor etc.)
- Devices can be found by querying the PCI controller and scanning the mapped devices though the nested laying of
    - Domain
    - Bus
    - Device
    - Function

PCI Layout

- PCI can be viewed as tree-like organization of I/O devices
- Each device mapped to PCI bus can be examined based on the IDs (device, vendor etc.)
- Devices can be found by querying the PCI controller and scanning the mapped devices though the nested laying of
  - Domain
  - Bus
  - Device
  - Function
- Linux kernel pre-creates this list and invokes the probe method of the matching driver when a driver is registered
- The "lspci" user space utility (and the /sys/bus/... interface) can be used to examine

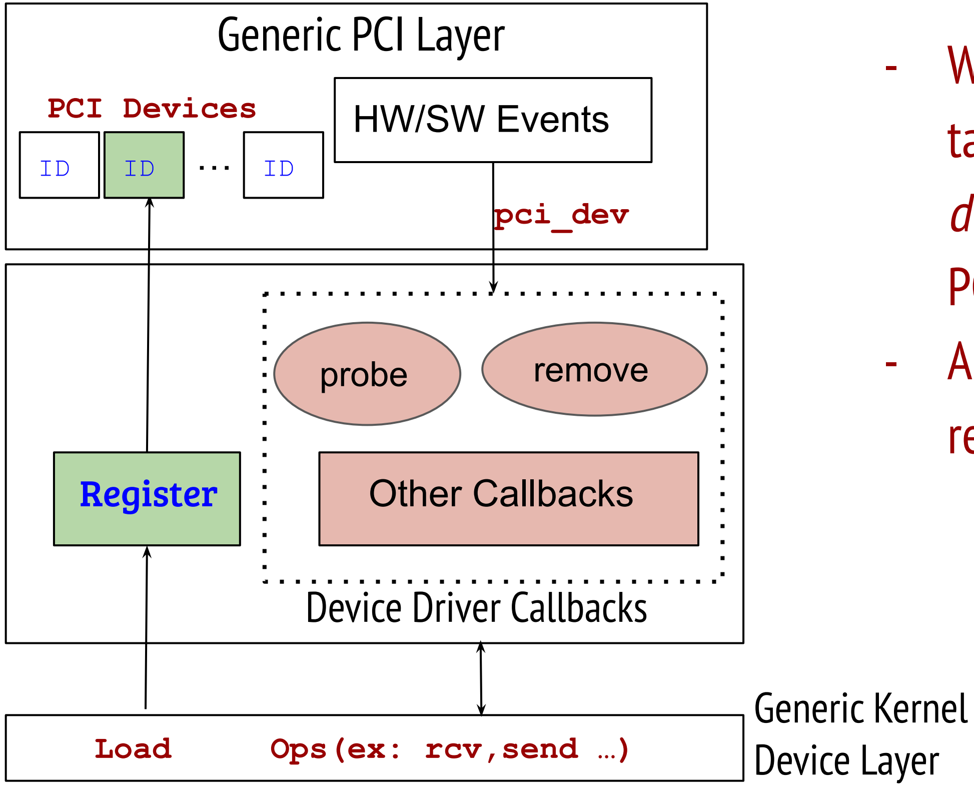
Linux PCI device driver

- A PCI device driver must register itself using an object of type “struct pci_driver”



Linux PCI device driver

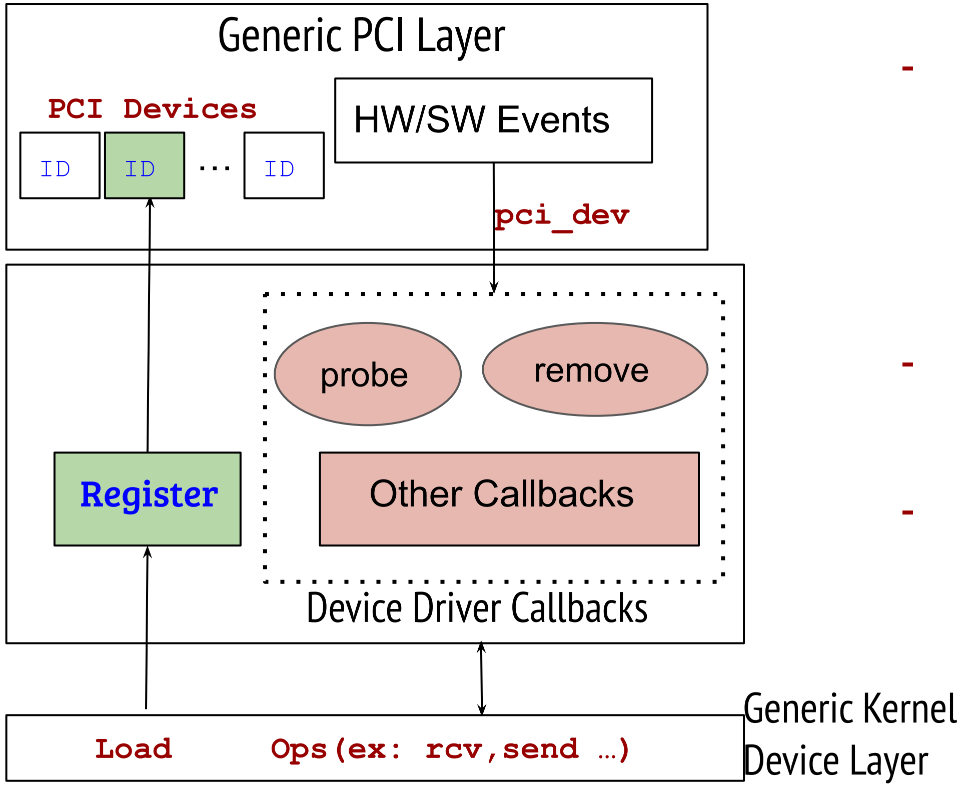
- A PCI device driver must register itself using an object of type “struct pci_driver”



- While registering a driver for a PCI device, an ID table containing a list of ID entries (*vendor, device, subvendor, subdevice*) are passed to the PCI layer to match a device for this driver
- A probe method (part of pci_driver structure) is registered as a call back

Linux PCI device driver

- A PCI device driver must register itself using an object of type "struct pci_driver"

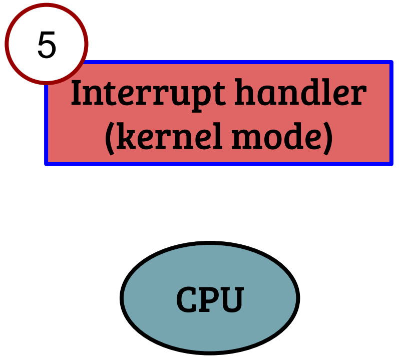


- While registering a driver for a PCI device, an ID table containing a list of ID entries (*vendor, device, subvendor, subdevice*) are passed to the PCI layer to match a device for this driver
- A probe method (part of pci_driver structure) is registered as a call back
- The generic PCI layer invokes the probe method to allow the device driver to perform device and software initializations (device API for the generic device layer)

Useful Kernel PCI helpers

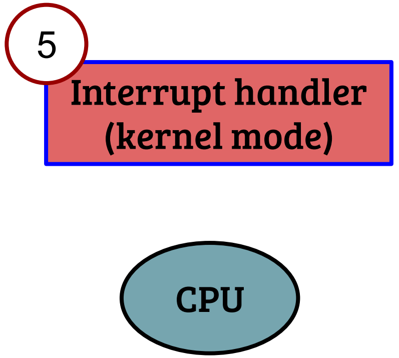
- Most PCI device drivers read and examine the BAR registers
- Reading the PCI configuration for any device (@PCI controller)
  - pci_read_config_byte/word/dword(pci_dev, offset, into)
  - pci_write_config_byte/word/dword(pci_dev, offset, from)
- Most PCI device drivers read and examine the BAR registers (BAR0, BAR1… Bar5)
  - pci_resource_flags(pci_dev, bar)
  - Type of resource (IO or MEM) can be examined, accordingly used for PIO or MMIO
  - pci_request_regions to check the I/O “address” resource availability
  - pci_resource_start(pci_dev, bar) returns handle to start of an I/O resource
- For MMIO resources
  - pci_ioremap_bar(pci_dev, barnum)
  - Returns a VA handle to operate on the device

Hardware interrupts (Background)



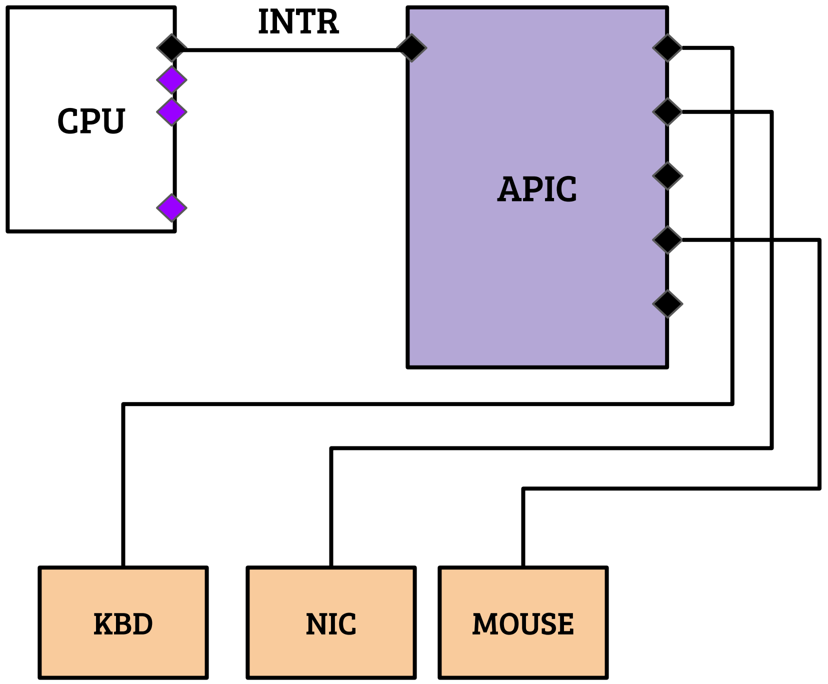
- Why interrupts?
- Example: Receive a packet from network
- What are the architectural support?

Hardware interrupts (Background)



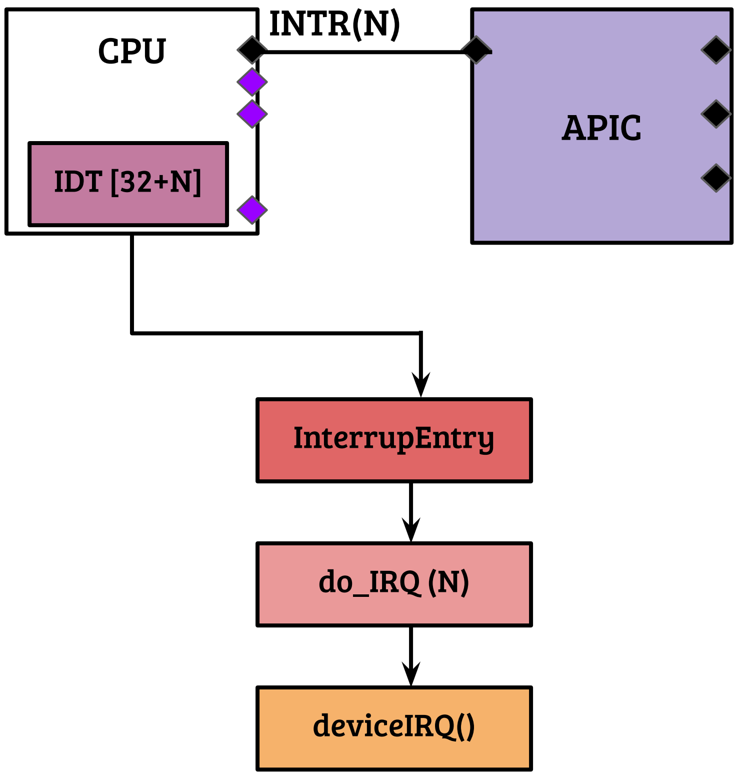
- Why interrupts?
- Example: Receive a packet from network
- Avoid CPU wastage due to polling
- Responsive and scalable systems
- What are the architectural support?
- CPU has limited #of interrupt PINs $\longrightarrow$ How to multiplex many devices?

Interrupt architecture - PIC and APIC



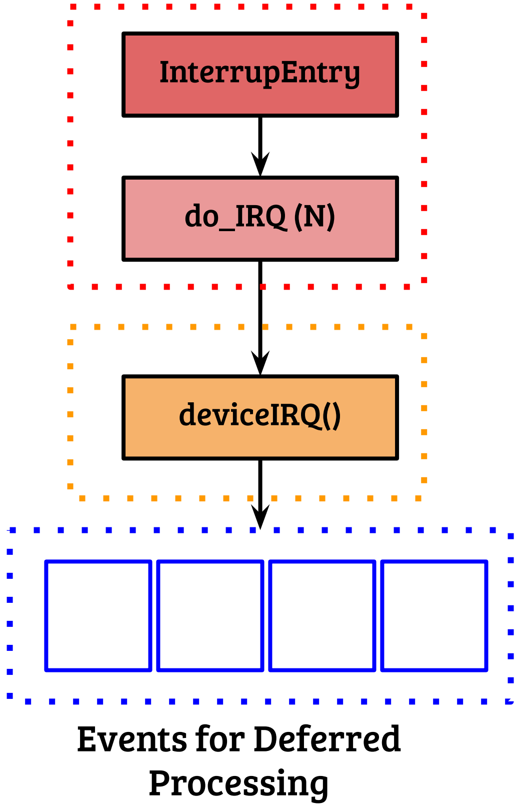
- Every device attached to the APIC is configured with a unique IRQ number
- APIC saves the IRQ in a control port register and raise CPU interrupt line on receipt of device interrupt
- CPU reads the IRQ number and invokes the interrupt handler
- Waits for acknowledgement before clearing the INTR line
- Selective disabling of IRQs possible
    - != cli (CPU interrupt disable)
    - New interrupts not lost

Interrupt handling



- IDT configured to load the interrupt execution context (CPL and stack)
- Interrupt entry: save regs, switch CR3 if needed
- do_IRQ checks the descriptor flags and invokes the real handler
- The device driver handler implements the device specific functionalities
- When is the interrupt acknowledged (i.e., INTR is cleared)?
- How long is the device interrupt masked?
- Not all interrupts can be handled quickly, e.g., NIC RCV

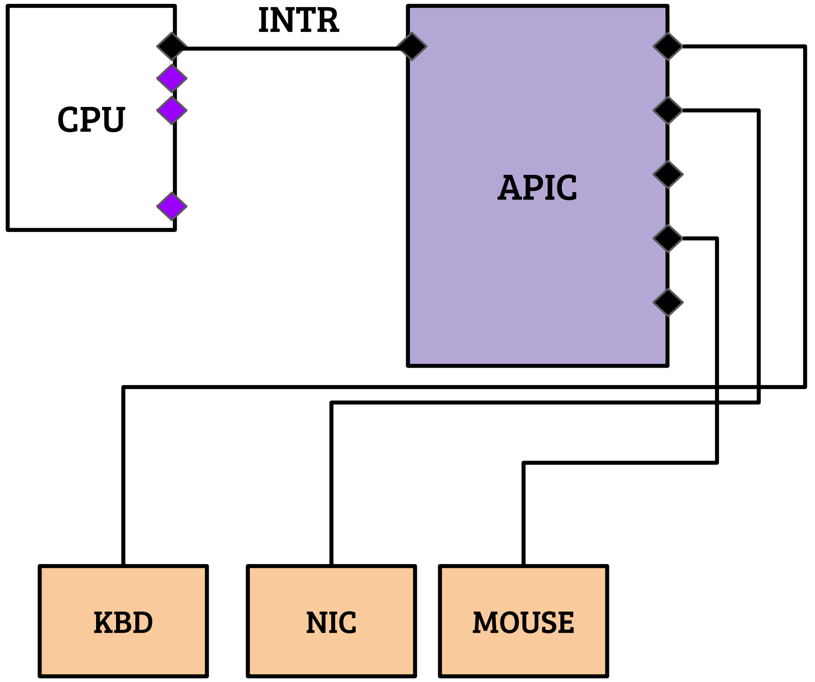
Interrupt handling in three stages



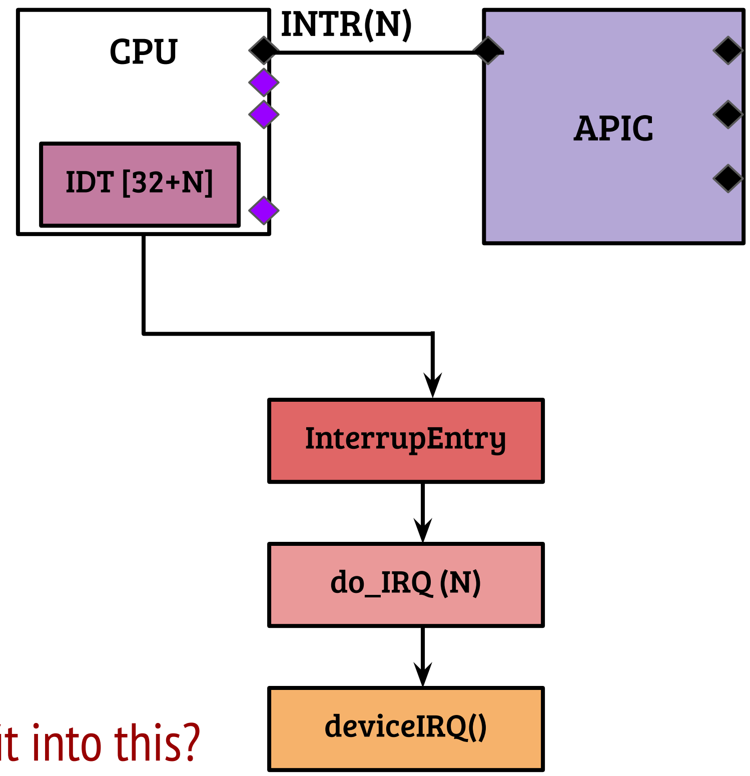
- Critical tasks: Interrupt context setup, APIC acknowledgement
- Semicritical: Accessing/updating device state, e.g., update receive queue pointers of a NIC
- Deferrable: Actions that are device independent e.g., Network stack processing

Interrupt handling

Interrupt Architecture

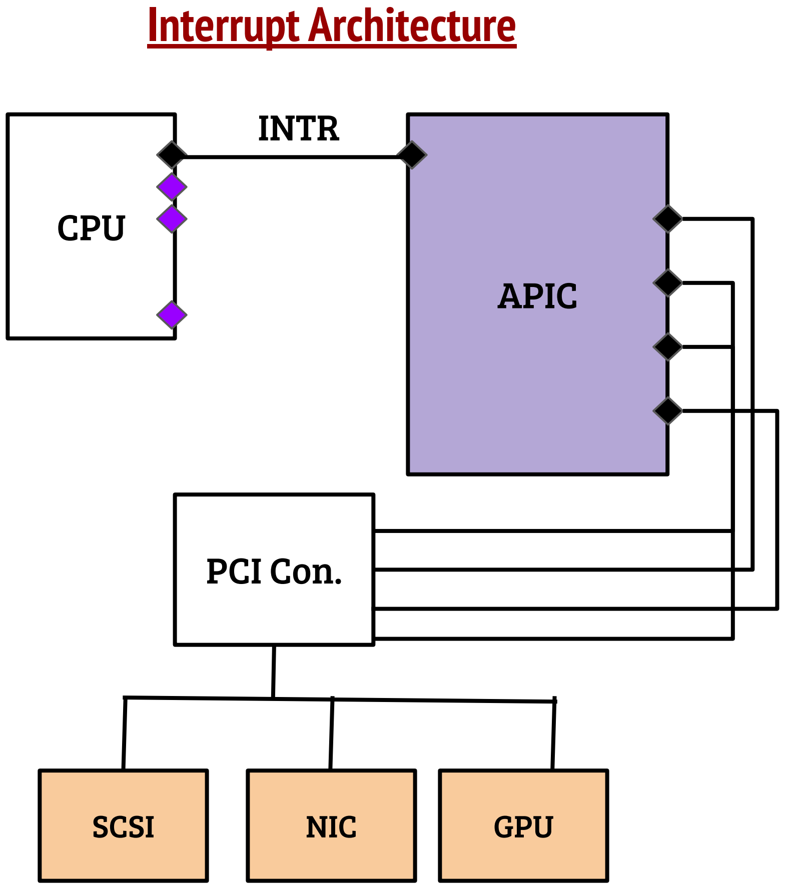


Software Interfacing



How does PCI fit into this?

PCI Interrupt handling



- How does PCI fit into this?
- A device connected through a PCI connector can use upto four interrupt PINs
- Each PIN can be independently forwarded to the core interrupt controllers (e.g, APIC or IOAPIC)
- Typically, IRQs are shared in PCI devices

Interrupts in PCI devices

- Examining interrupt capability
  - Reading the PCI config using pci_read_config_byte (IRQ pin and IRQ line) directly
  - Using the PCI helper APIs such as
    - pci_alloc_irq_vectors
    - pci_irq_vector
    - request_irq

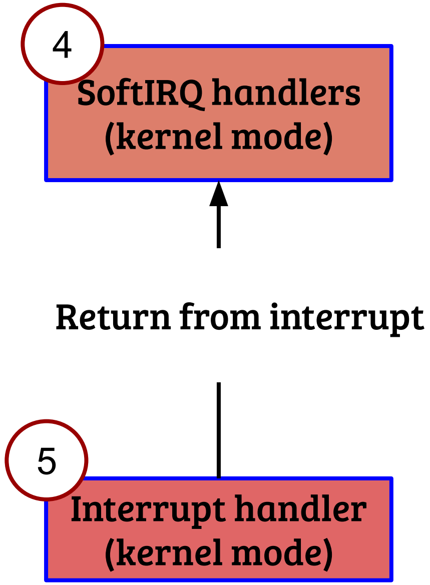
Interrupts in PCI devices

- Examining interrupt capability
  - Reading the PCI config using pci_read_config_byte (IRQ pin and IRQ line) directly
  - Using the PCI helper APIs such as
    - pci_alloc_irq_vectors
    - pci_irq_vector
    - request_irq
- Interrupt handler
  - The device level callback for interrupt handling is registered during request_irq
  - The handler code must determine if the IRQ belongs to the device, why? And How?

Interrupts in PCI devices

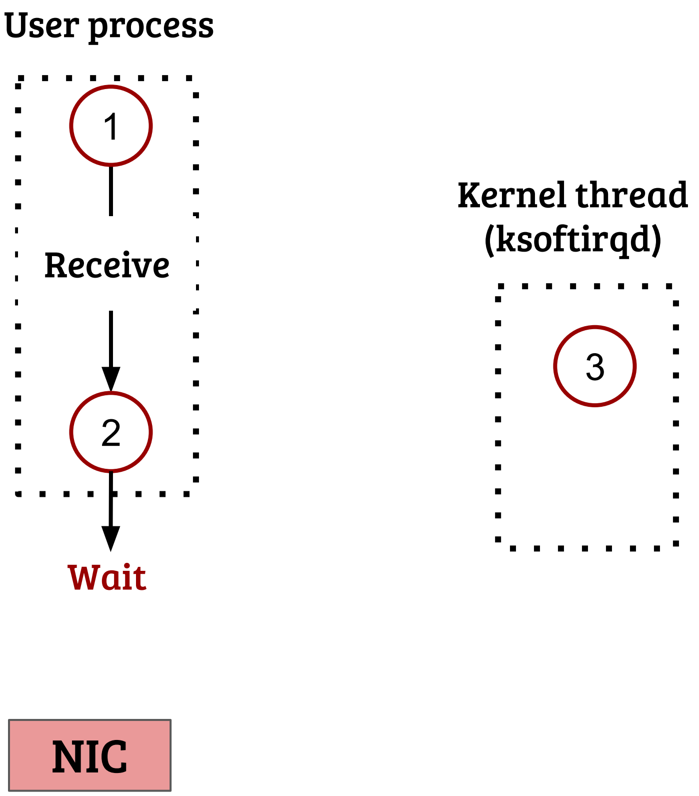
- Examining interrupt capability
  - Reading the PCI config using pci_read_config_byte (IRQ pin and IRQ line) directly
  - Using the PCI helper APIs such as
    - pci_alloc_irq_vectors
    - pci_irq_vector
    - request_irq
- Interrupt handler
  - The device level callback for interrupt handling is registered during request_irq
  - The handler code must determine if the IRQ belongs to the device, why? And How?
    - IRQ may be shared across many devices
    - By checking the interrupt status register of the device

Interrupt handling: SoftIRQ


4
SoftIRQ handlers (kernel mode)
Return from interrupt
5
Interrupt handler (kernel mode)

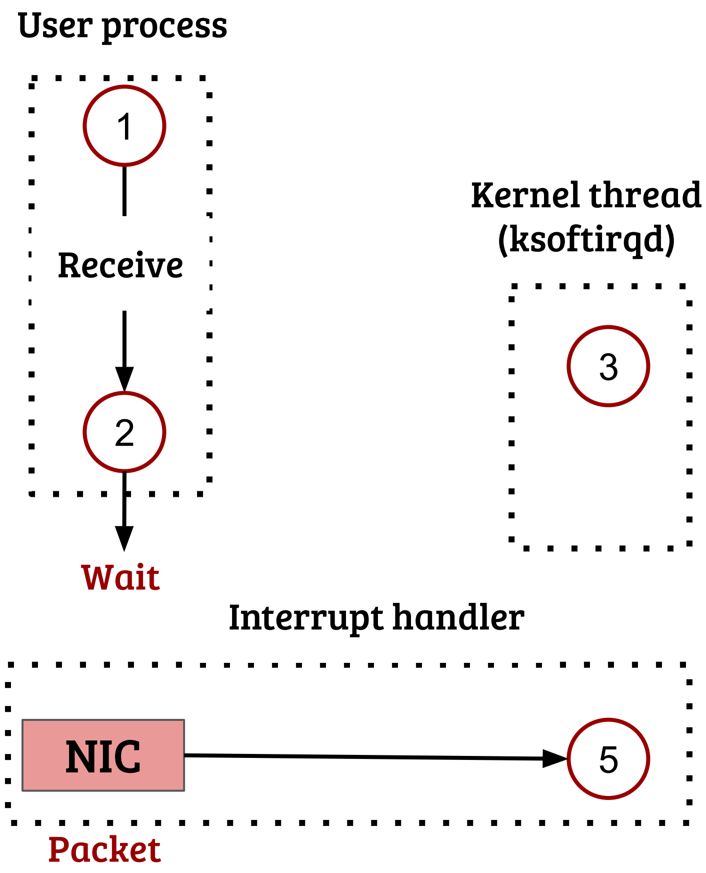
- Carry out deferrable operations, can be preempted by interrupts
- Like an interrupt, it can be raised, disabled, enabled, masked
- Executed by the local CPU kernel thread (*ksoftirqd*, one per CPU)
    - Infinite loop checking for pending softIRQ (set when softirq is raised)
    - Often scheduled on irq_exit( ) or explicit wakeup

Contexts in action: network receive



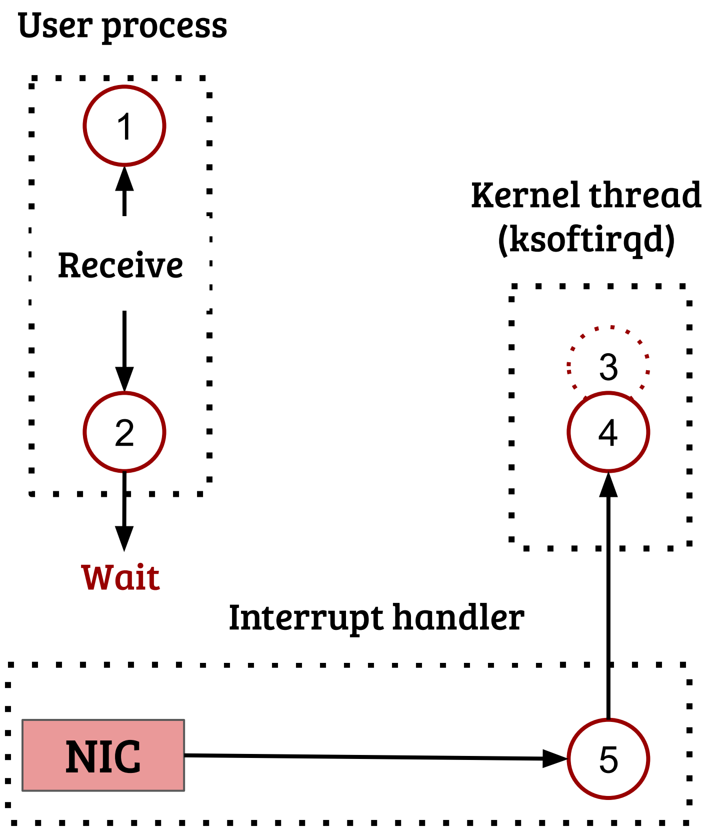
- The user process invokes recv( ) system call (blocking)
- No processed payload found, the process is descheduled and put into a wait queue
- Ksoftirqd is either suspended or processing other pending softIRQs

Contexts in action: network receive



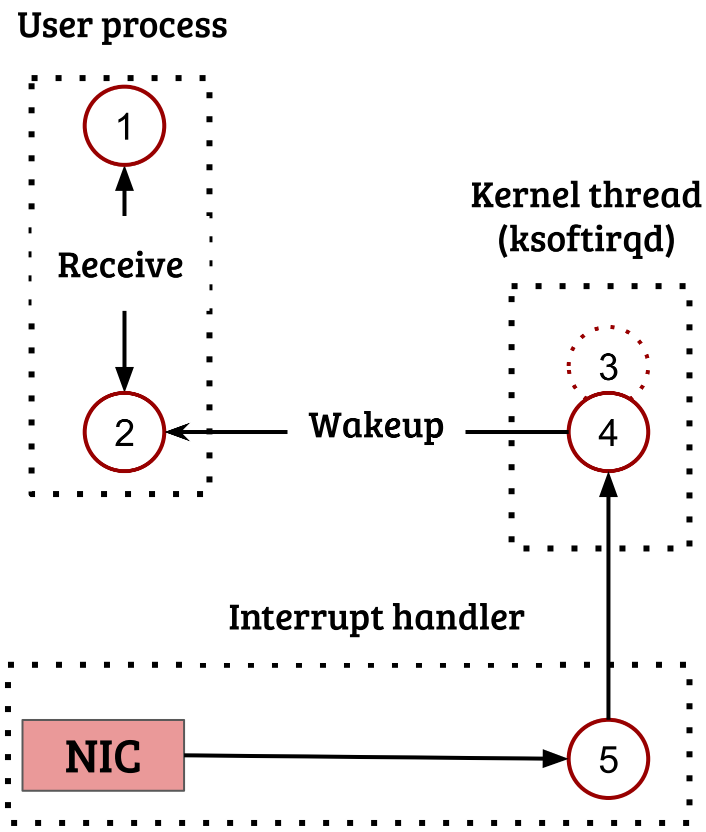
- The NIC copies the packet (using DMA) into memory buffers (a.k.a. skbuffs) and triggers the interrupt
- Before the device specific interrupt handling, APIC is acknowledged
- The device interrupt handler update the device state while masking device interrupts
- Queues the packet for further processing and triggers a softIRQ

Contexts in action: network receive



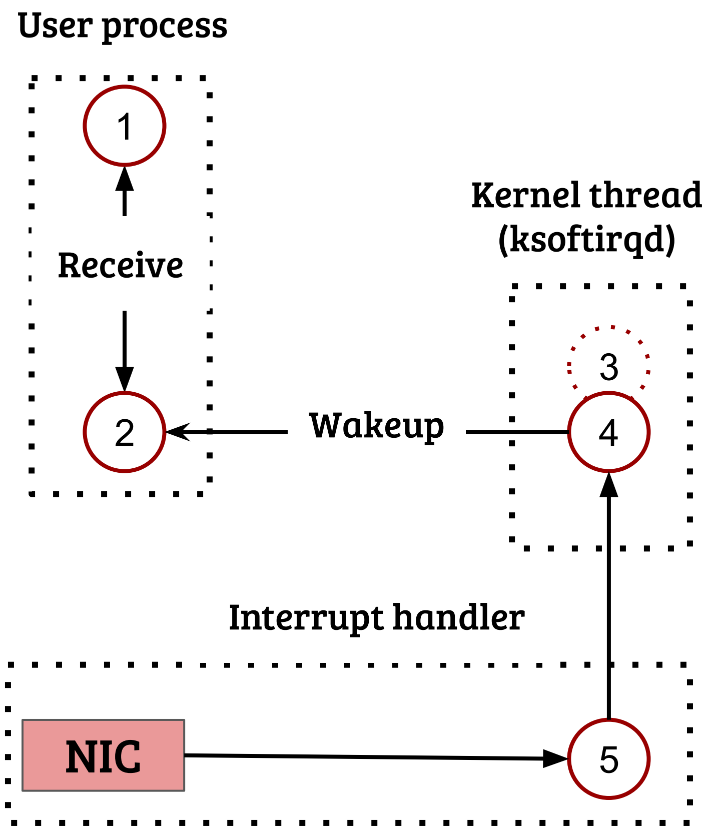
- The softIRQ is scheduled using the ksoftirqd kernel thread context
- Protocol stack processing is performed in this context
- As part of the protocol processing, the destination process is derived

Contexts in action: network receive



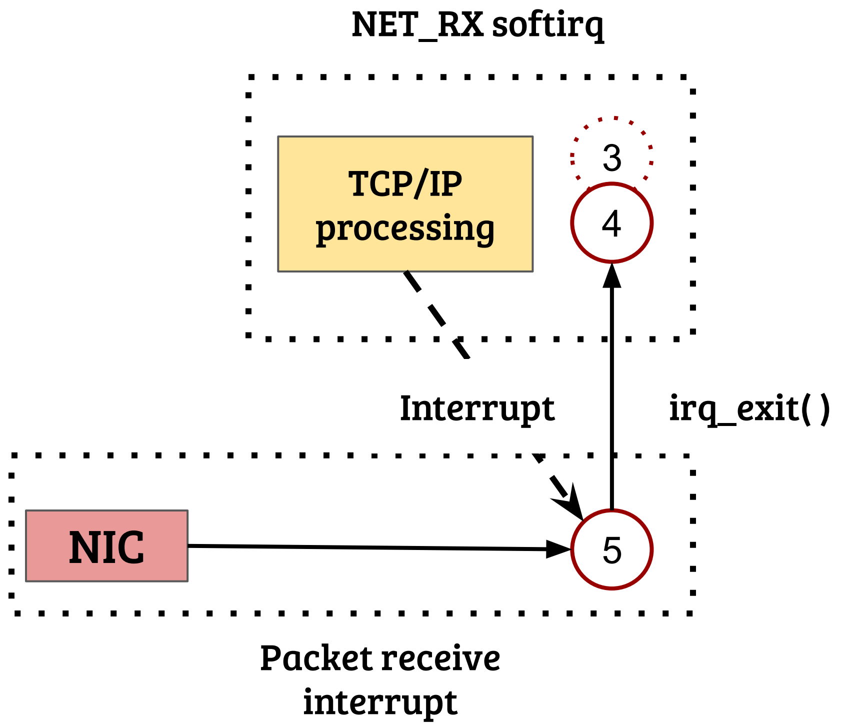
- The softIRQ processing wakes up the user process
- The user process returns from syscall (copy payload to user)
- Now, what could be the issues with this approach?

Challenges in network receive



- Minimize network packet copy across the contexts
- Precise scheduling: application progress and fairness
- Network is always overdriven and self-adjusting in nature → rate limit as early as possible
- Issues
    - Receive livelock: CPU is always handling interrupts
    - User process starvation due to softIRQ processing

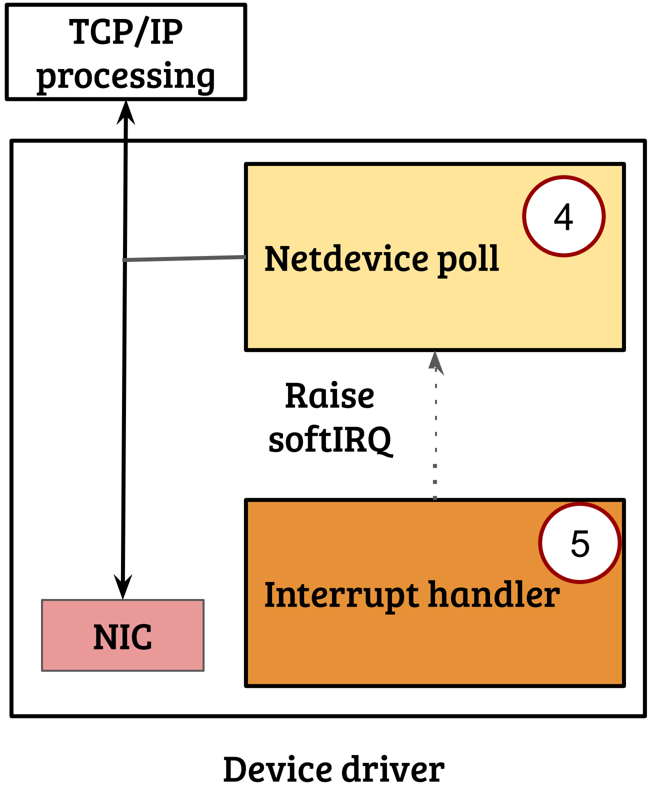
Receive livelock [1]



- Root cause: Interrupts have the highest priority over other contexts
- If the rate of interrupts is high, the system remains in interrupt handling mode, resulting in *receive livelock*
- Solution approach: Lower the priority of interrupts under heavy load
- How?

1. https://www.usenix.org/legacy/publications/library/proceedings/sd96/mogul.html

NAPI: Interrupt + Polling



- Interrupt handler raises softIRQ after disabling packet receive interrupts
- Driver registered poll method is invoked
  - Executes till receive queue is empty or an upper threshold (budget)
  - Enable the interrupt (if queue is empty) and return
- Advantages
  - Low network load, more interrupt driven
  - High load, less interrupt processing
  - Avoid wasted work, drop packets early (in the device buffer)