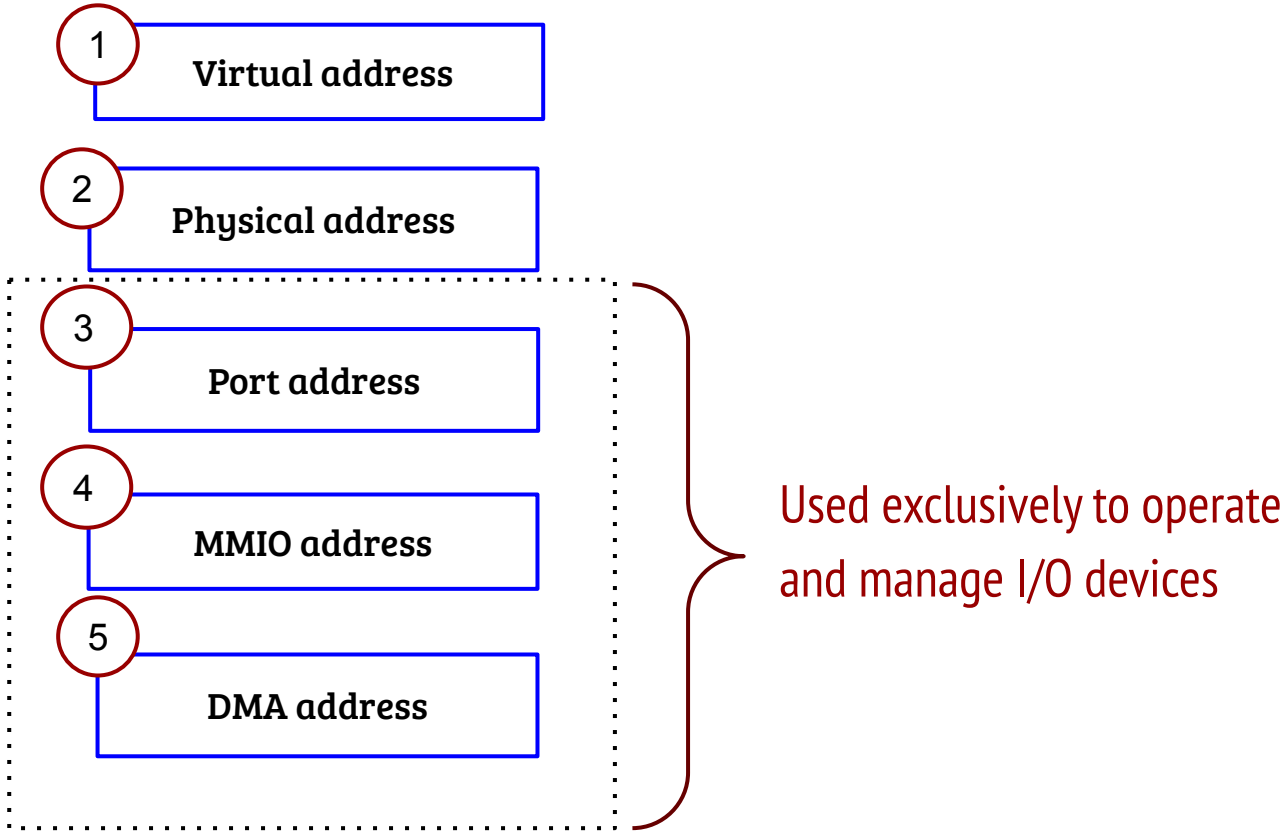
# CS614: Linux Kernel Programming

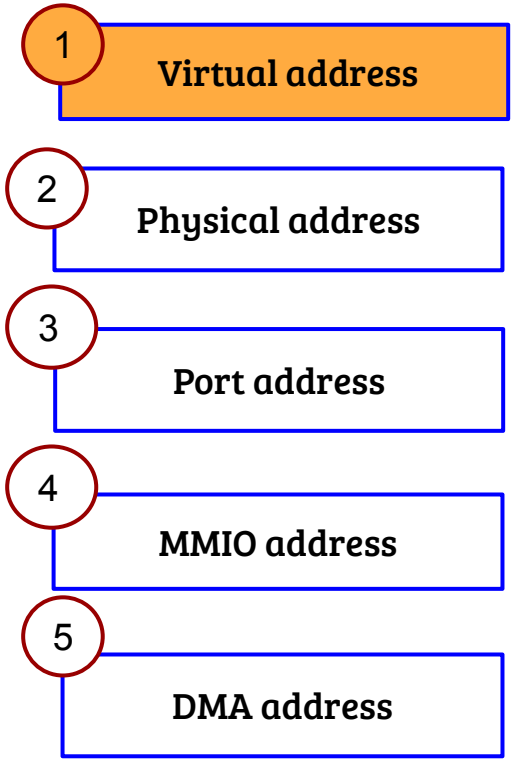
## I/O Addressing in Linux Kernel

Debadatta Mishra, CSE, IIT Kanpur

# Address types in kernel

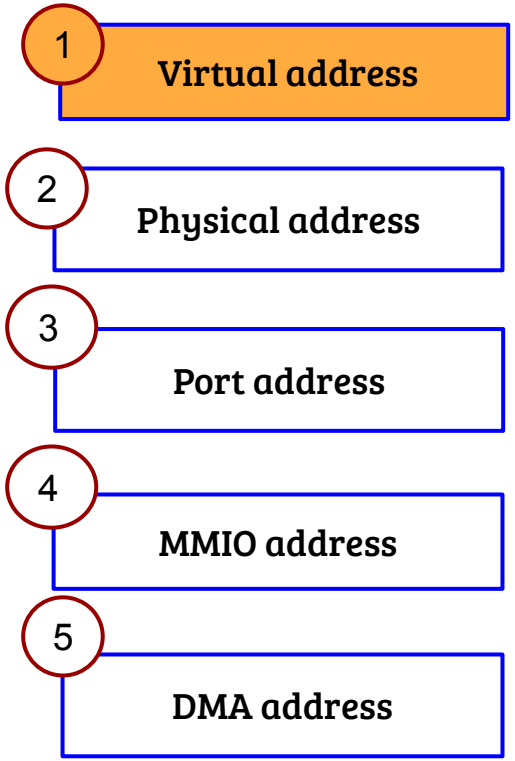


# Kernel virtual address



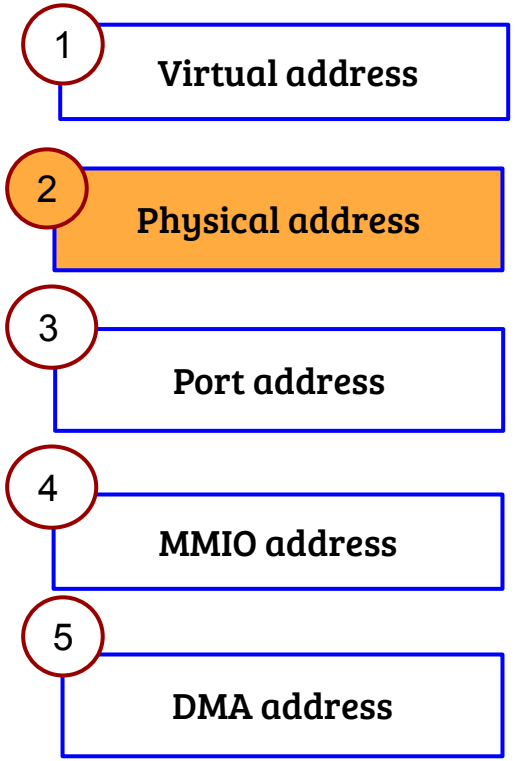

- Direct mapping of physical memory (64TB)
  - Conversion from virtual to physical and vice-a-versa can be done using macros like __va(paddr) and __pa(vaddr)

# Kernel virtual address



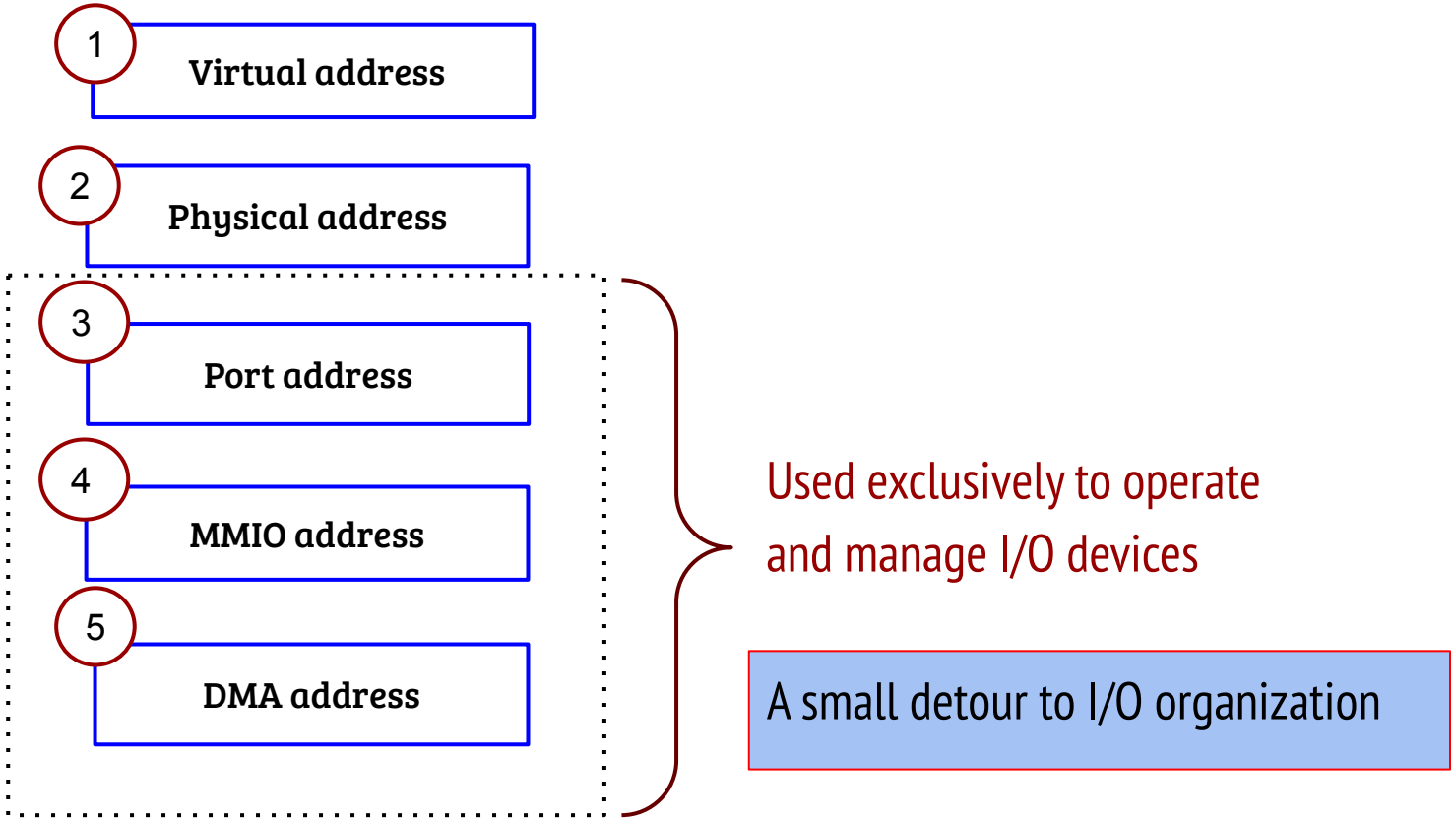

- Direct mapping of physical memory (64TB)
  - Conversion from virtual to physical and vice-a-versa can be done using macros like __va(paddr) and __pa(vaddr)
- Physically discontinuous virtual address
  - Allocated used vmalloc( )
  - Useful when you allocate large contiguous kernel virtual address
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

# Physical address in kernel

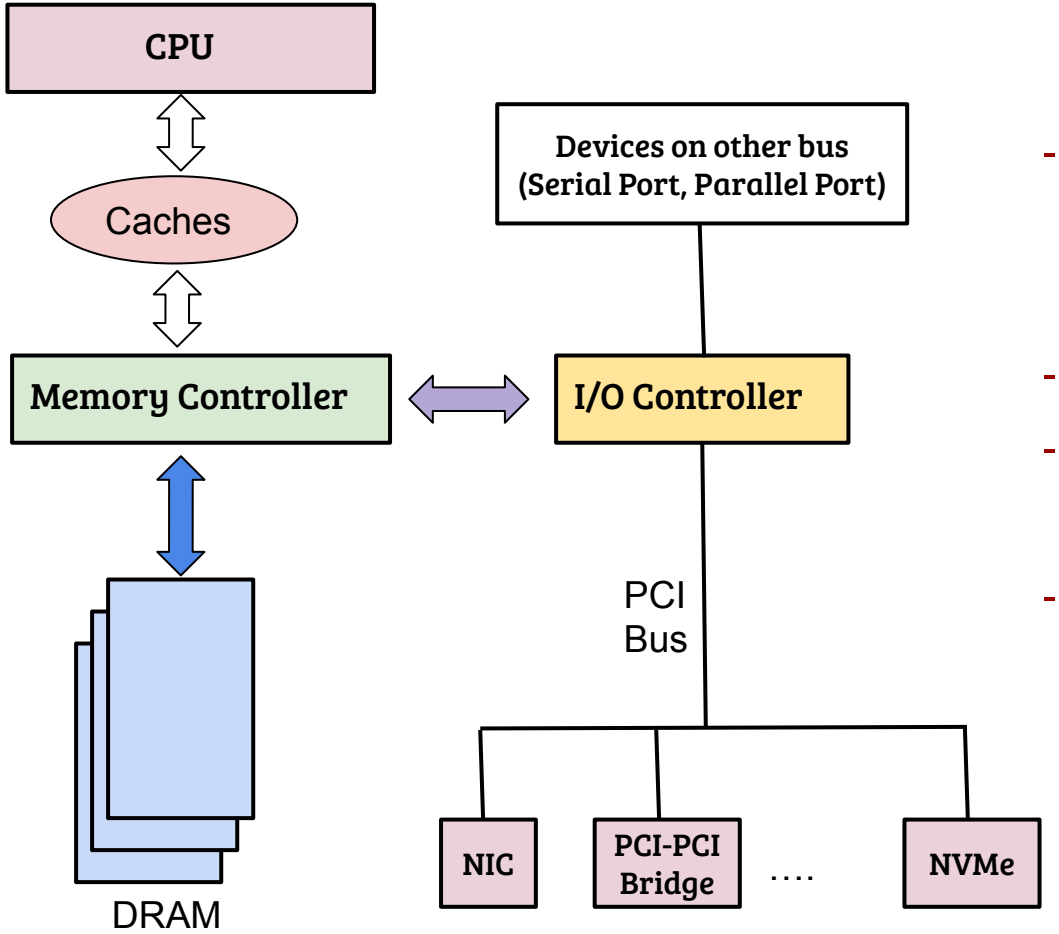



- Two commonly used (almost interchangeable) terms
  - Page: A *struct page* type
  - Physical Frame Number (PFN): *unsigned long*
  - APIs: pfn_to_page, page_to_pfn etc.
  - How does the conversion happen?
- At the lowest level, physical allocation done through page allocation APIs (alloc_page, free_page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

# Address types in kernel

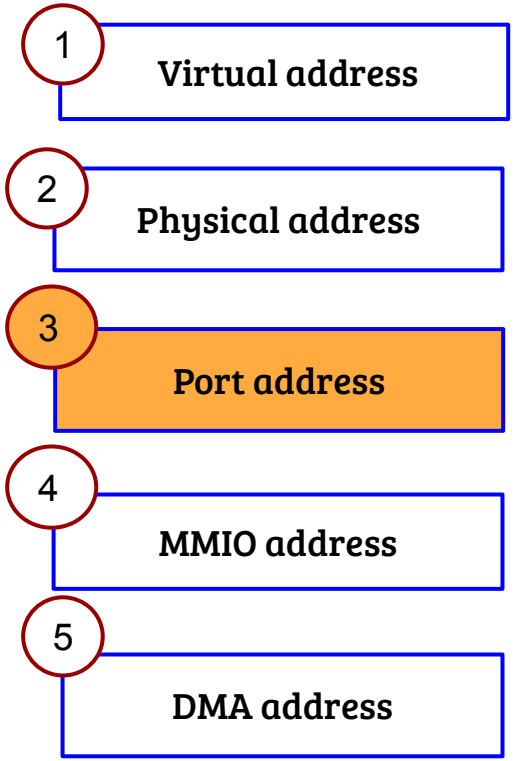


# I/O device interfacing (example organization)




- 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?

# 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” ⟶ 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?

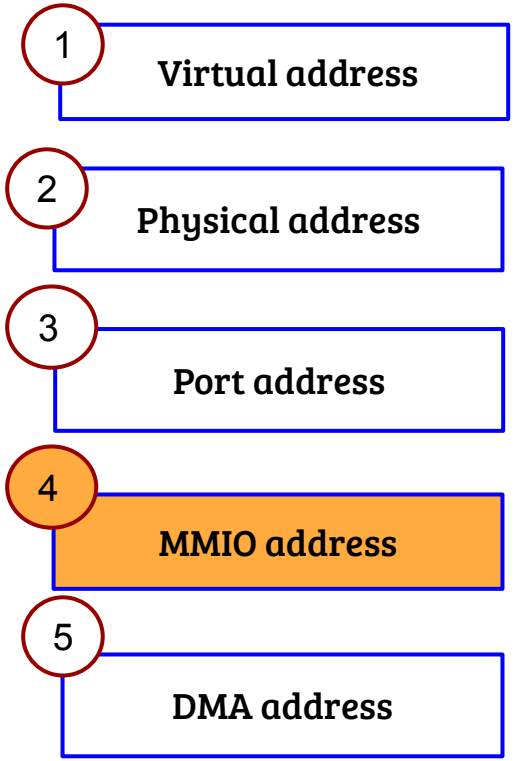
# Port I/O access

- 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

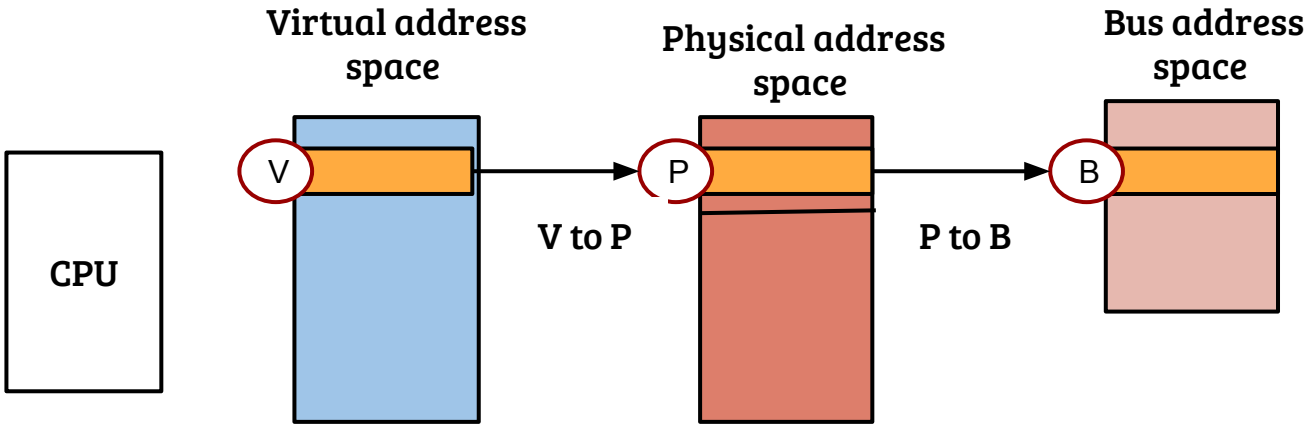
- Instructions: inb, outb, inw, outw, inl, outl
- Example: “outb $0x3F8, $0x5” ⟶ Write five to the port address 0x3F8
- Important: Completion of a write instruction may not imply the intended I/O operation is completed (CPU and I/O speeds may not match!)
- Example: To print a string into a serial console 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



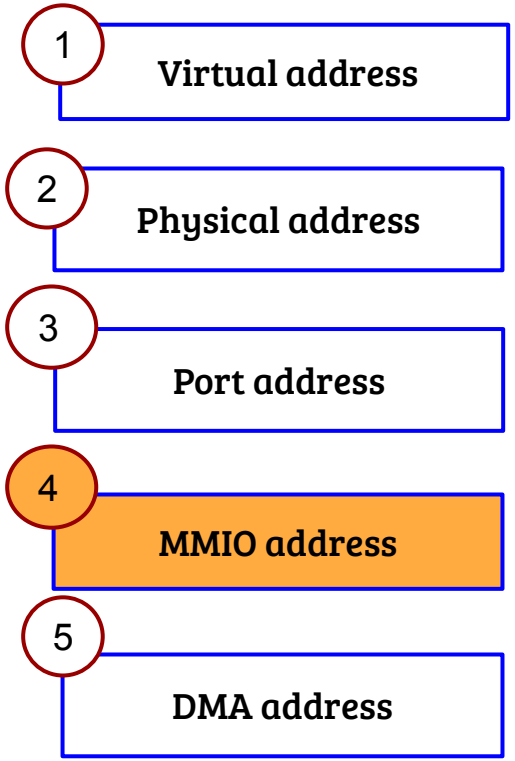

- I/O registers/memory mapped into physical address space, can be accessed like memory
- What address to use, virtual or physical?
- What extra care to be taken while accessing MMIO addresses?

# 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




- 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?

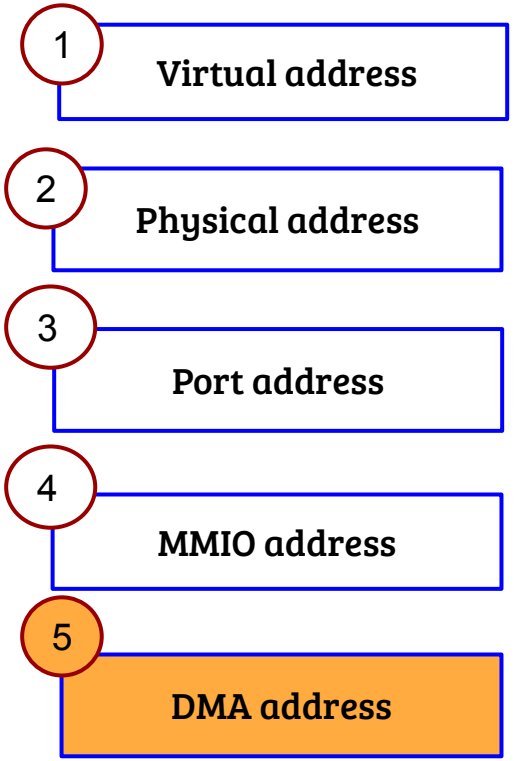
# PIO and MMIO: User mode vs. Kernel mode

- 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

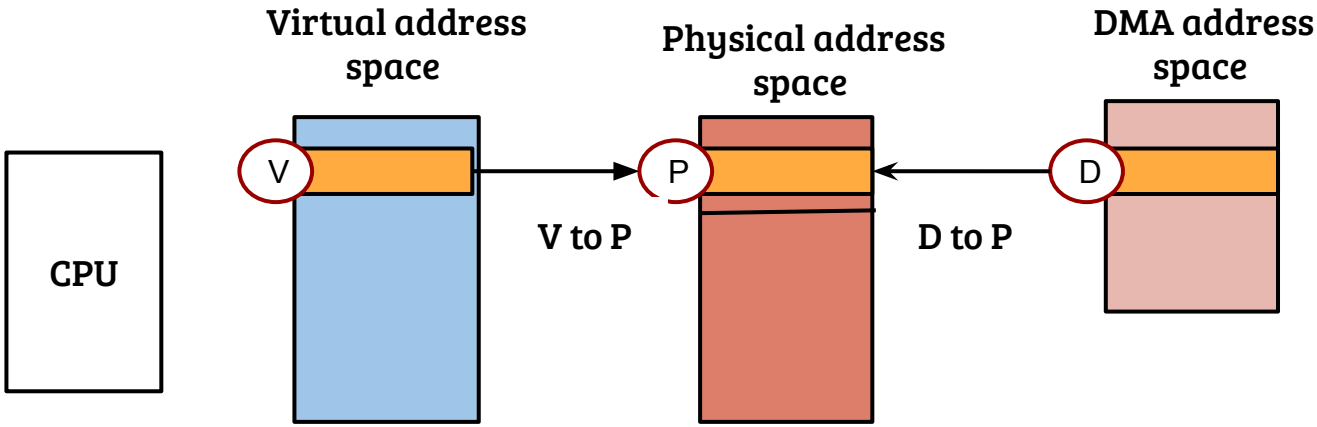
- Isolation requirements require I/O access restrictions from the user space
- However, in some cases, it may be required; Can the OS allow I/O access from user mode?
- Port I/O?
  - In intel X86 systems, IOPL bit in the flags register can be used to control access
  - For finer granularity control,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




- 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.

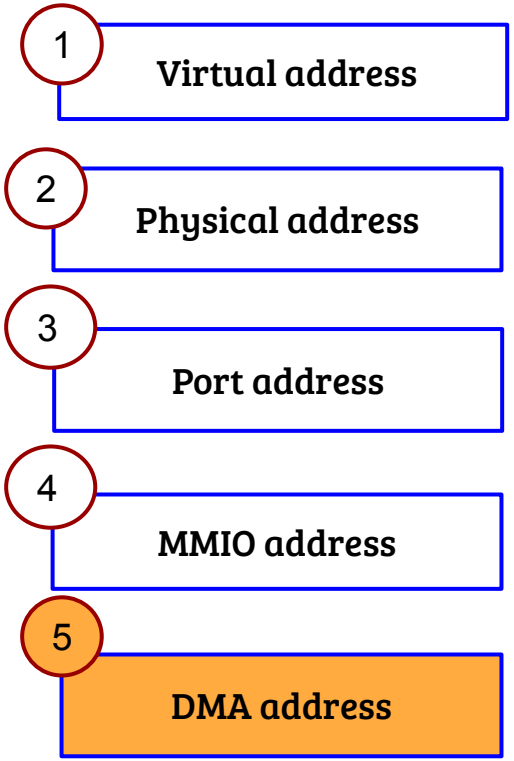



- Device driver allocates a buffer (VA = V, PA = P), no lazy allocation allowed!
- In non-IOMMU systems, device can use P directly
- With IOMMU, mapping must be setup between D ⟶ P using API's like *dma_map_single*
- Why device driver programmer has to worry about the DMA address?

# DMA and interrupt handling example

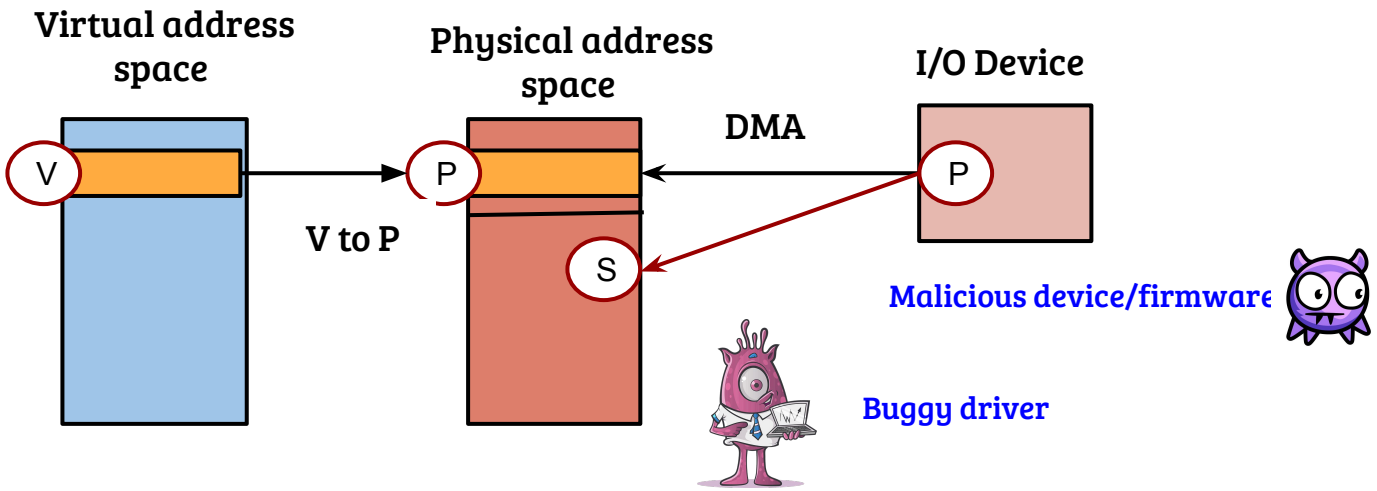
```
setup_one_rcv(NIC *nic){
    dma_addr_t *mapping;
    mapping = dma_map_single(nic->dev, nic->buff_va, nic-> len, DMA_FROM_DEVICE);
    nic->rcv_dma = mapping;
    mmio_nic(nic, DEVICE_SET_DMA);
}

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

# Direct memory access (DMA)



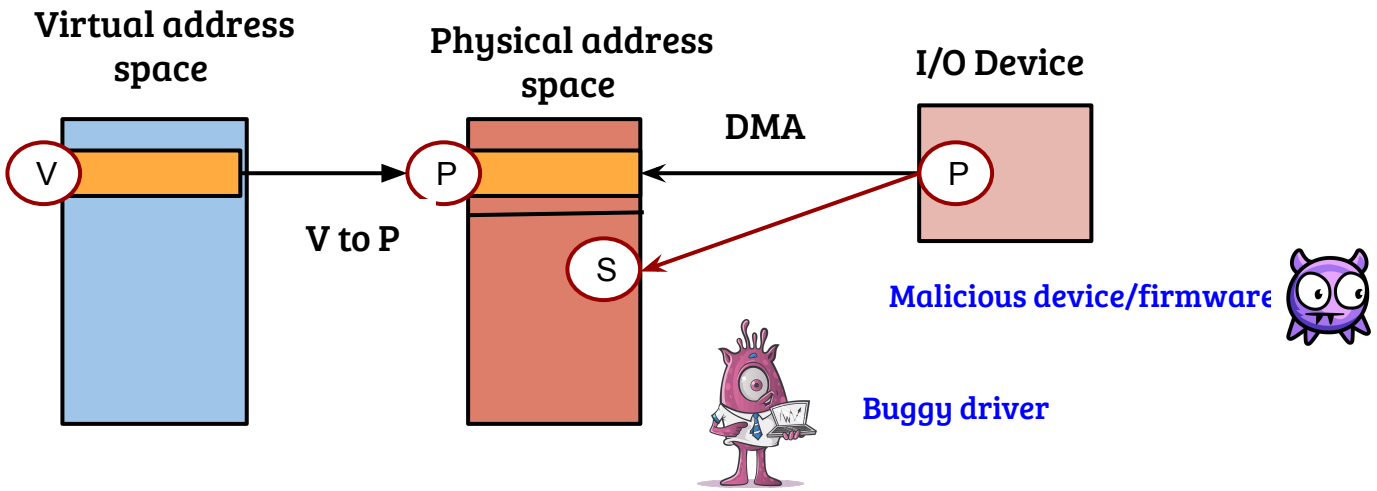

- Virtual addresses used by DMA should be mapped (don't use vmalloc( ) address)
- DMA mapping can be of two types
  - Consistent/Coherent: mostly used throughout the driver lifetime
  - Streaming/inconsistent: used to configure receive buffer of a NIC
- Refer to kernel documentation (Documentation/core-api/dma-api-howto.txt)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

# Security issue with DMA



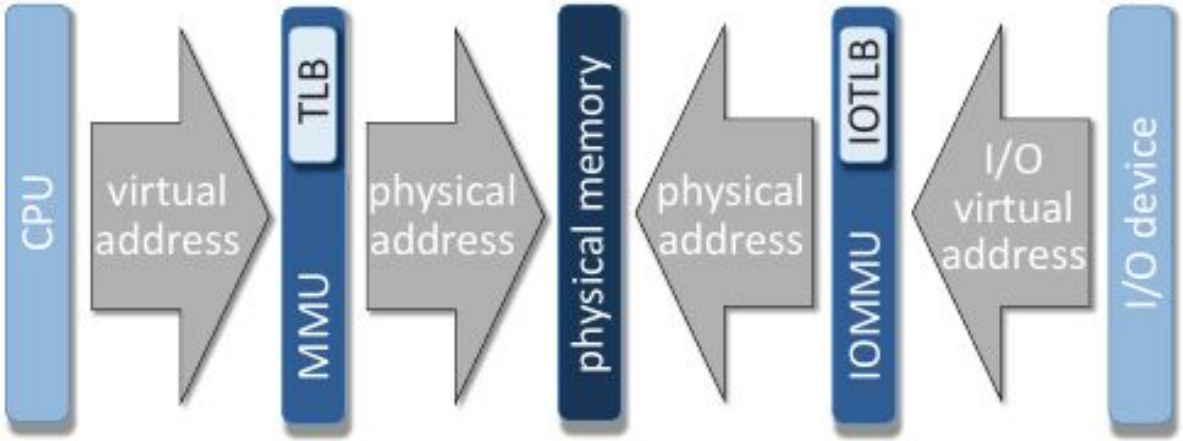

- I/O devices can access arbitrary memory locations
- Compromised security, information disclosure
- How to address this issue?

# 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