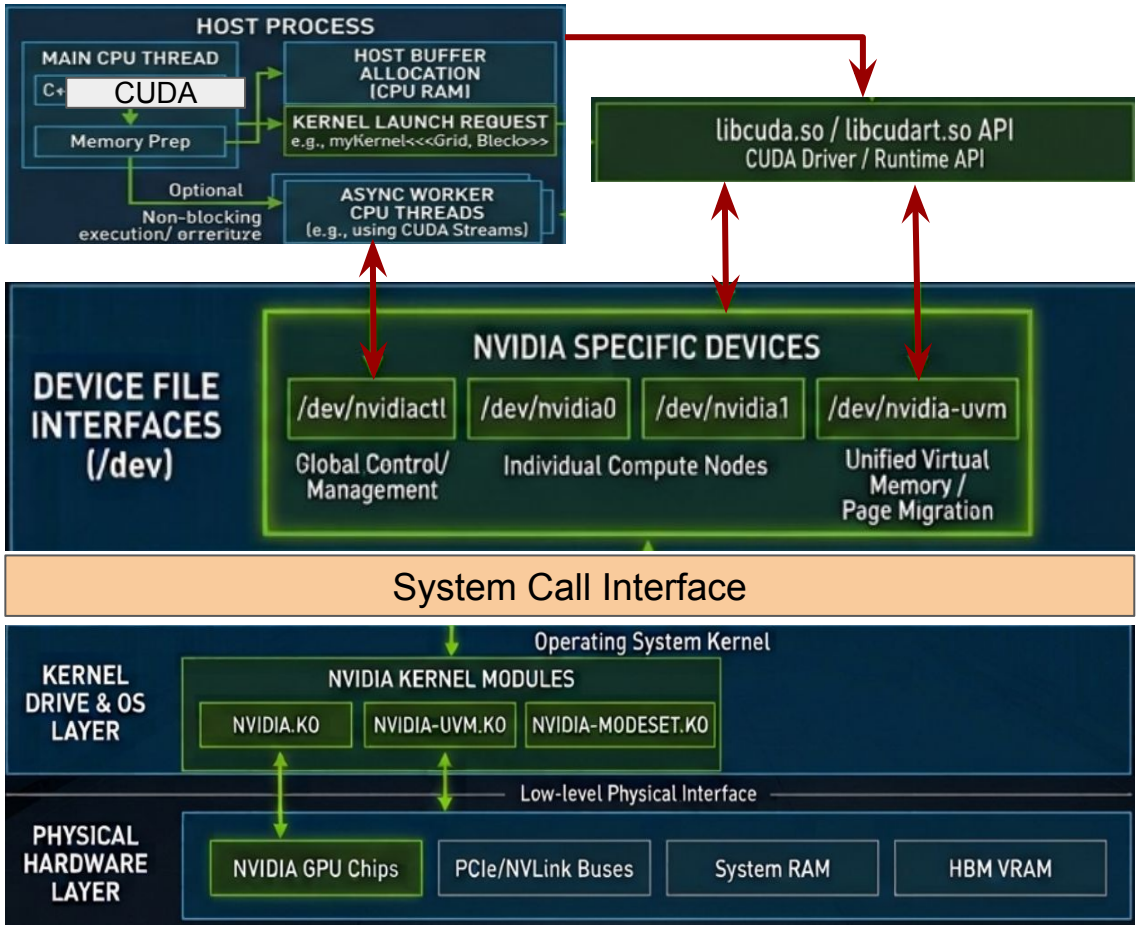


IIT KANPUR
Indian Institute of Technology Kanpur

CS623: GPU Architecture and Programming

Software Stack

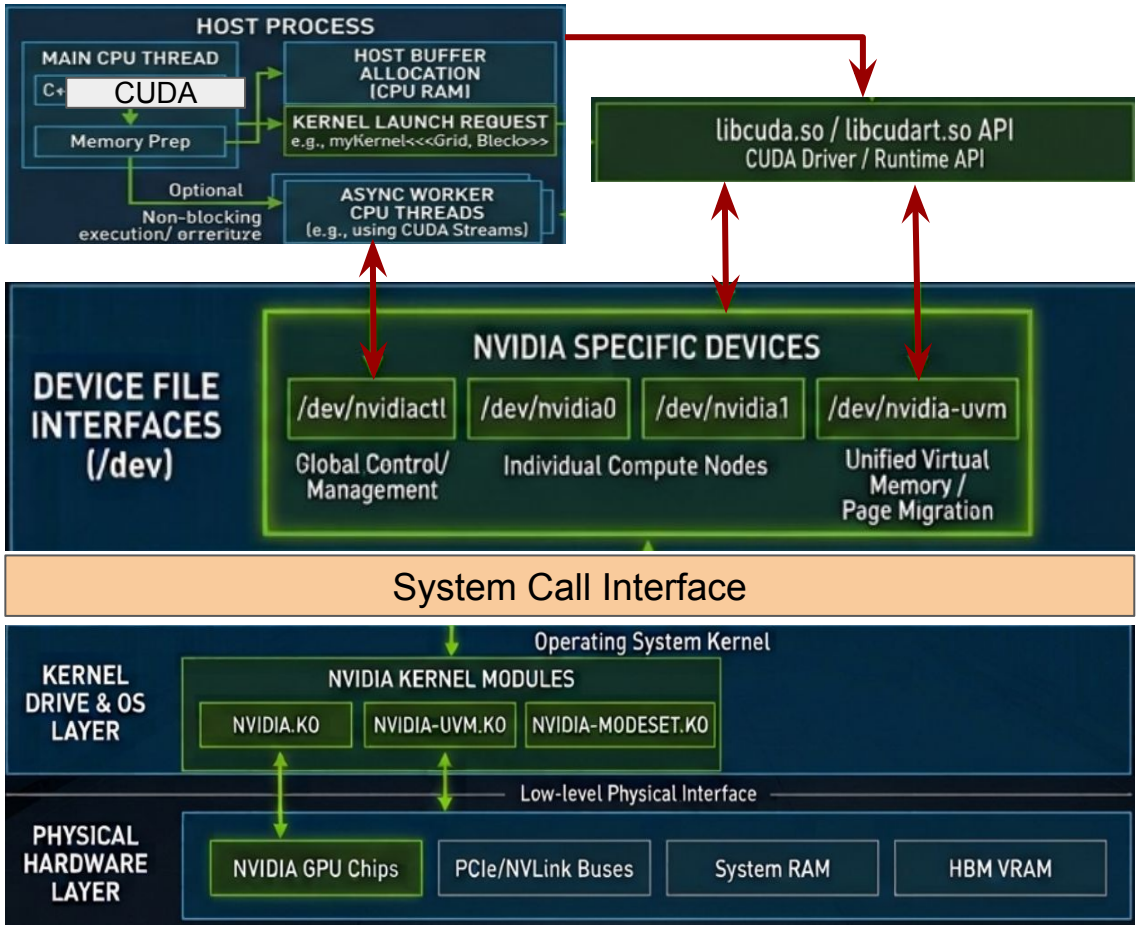
Software Stack: Big Picture*



- Cuda program becomes an OS process on execution
- The process is multi-threaded
  - Async. notification etc.
- The CUDA library provides most of the user-space helpers—allocation, kernel launch etc. using system calls
- Heavily used subsystems: POSIX file I/O, memory management, threads and event synchronization

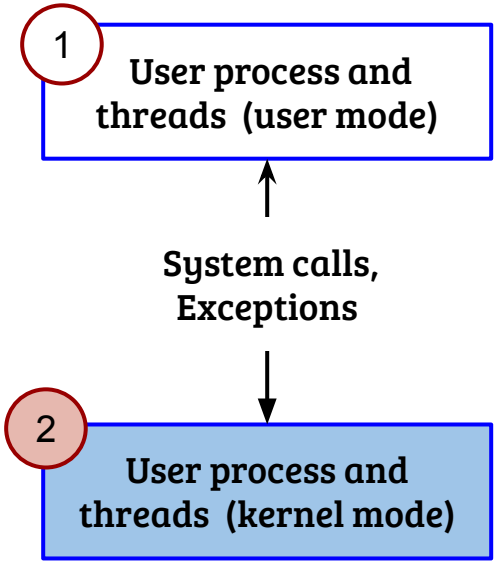
*Parts of the Figure are AI generated

Software Stack: Digging a little

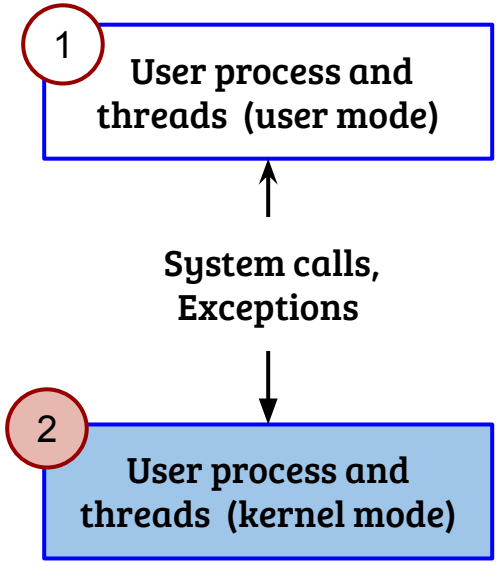


- The device driver modules (in kernel) provide device interfaces through devfs
- Everything is a file in Linux (Unix), Thanks to Richie and Thompson!
- *How device files interface with user space to perform operations on physical devices? What is the role of the OS (e.g., Linux)?*

User contexts: user mode and kernel mode

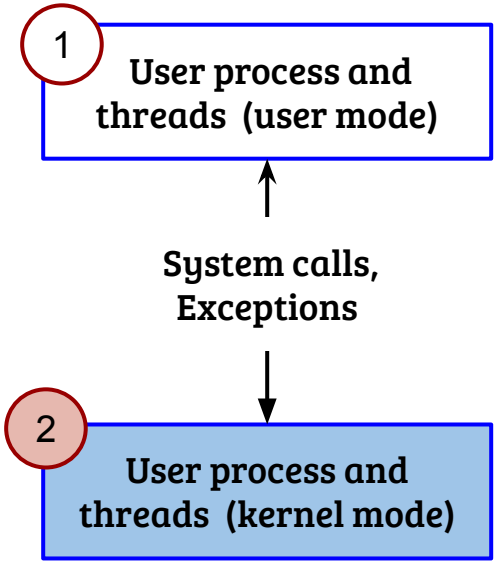


User contexts: user mode and kernel mode



- What are the differences in execution states?
- What is the exact entry and exit mechanisms—user to kernel context switch and vice-a-versa?
- What is the need for save and restore of the execution states? How implemented in Linux kernel?
- Access/modification of user execution state from the kernel mode, how?
- How OS manages the user contexts?

User contexts

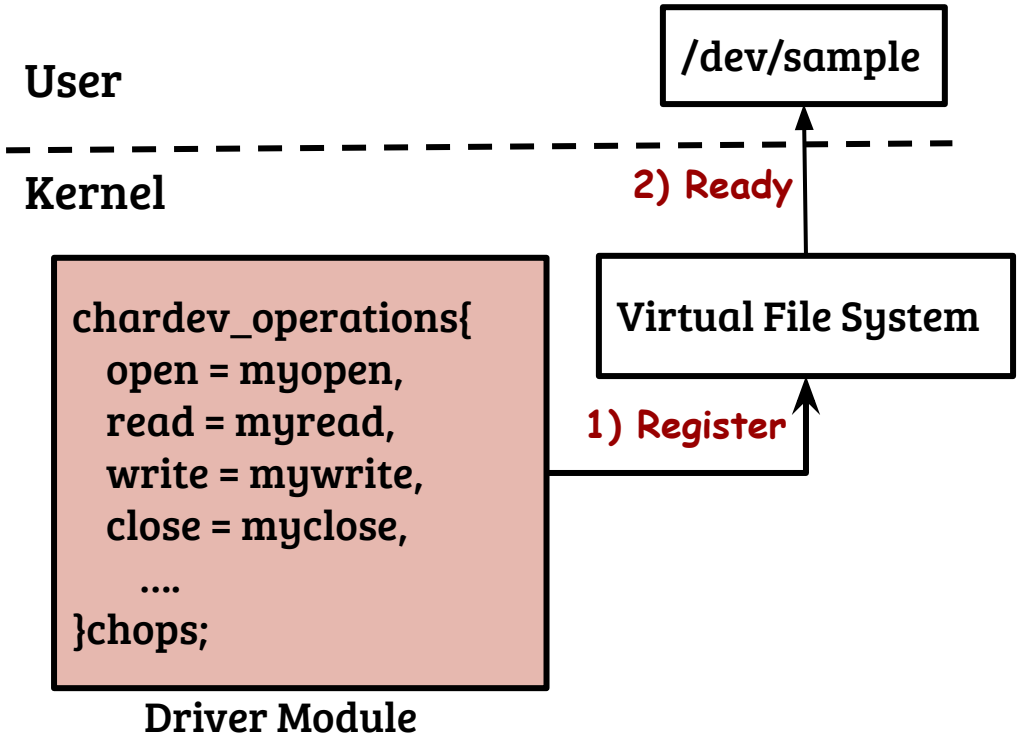


- What are the differences in execution states? Priv., stack, address space (partially or completely)
- What is the exact entry and exit mechanisms—user to kernel context switch and vice-a-versa? Handshake with hardware—IDT and syscall entry configurations
- What is the need for save and restore of the execution states? How implemented in Linux kernel? Correct execution, saved using the kernel stack
- Access/modification of user execution state from the kernel mode, how? Can be accessed from the kernel stack (memory state access require special care TBD)
- How the kernel manages the user contexts?

Representing User Execution

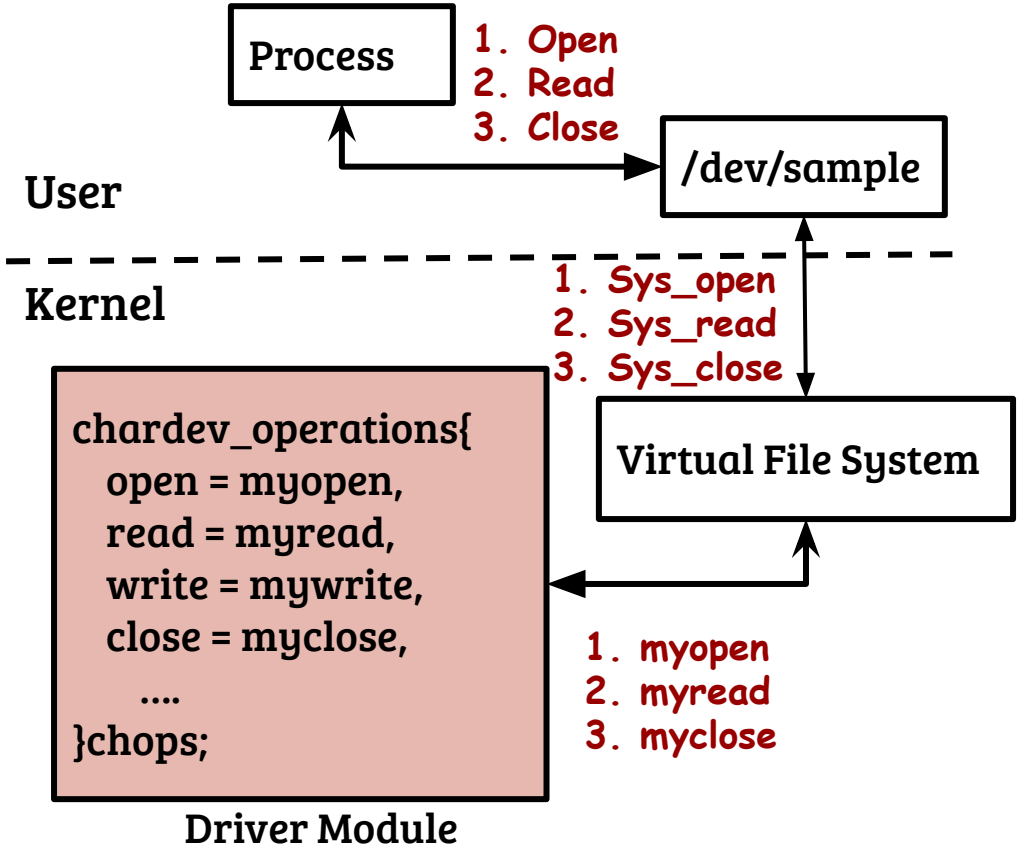
- Linux maintains one instance of the *task_struct* (defined in *include/linux/sched.h*) for each user process (and thread)
- Each task struct is identified by a unique `pid'
  - How user process and threads are distinguished? (using tgid, TBD)
- Some of the important sub-system states
  - Address space ⟶ *struct mm_struct *mm;*
  - Open file information (includes device files, pipes) ⟶ *struct files_struct *files;*
  - Signal handling information ⟶ signal_struct, sigpending etc.
- Many other members for different configurations and settings (will be examined on-demand during the lectures)

User kernel interfacing with chardev



- Module must provide methods to handle operations on the chardev
- Everything is a file in Linux (Unix), Thanks to Richie and Thompson!
- Now the device is ready to be used! But how?

User kernel interfacing with chardev



- User process can use system calls like open( ), read( ), ioctl( ) etc.
- Just like normal file read/write, data can be passed to and from the device
- The module implementation determines the exact behavior of system calls through actual device interactions (if needed)

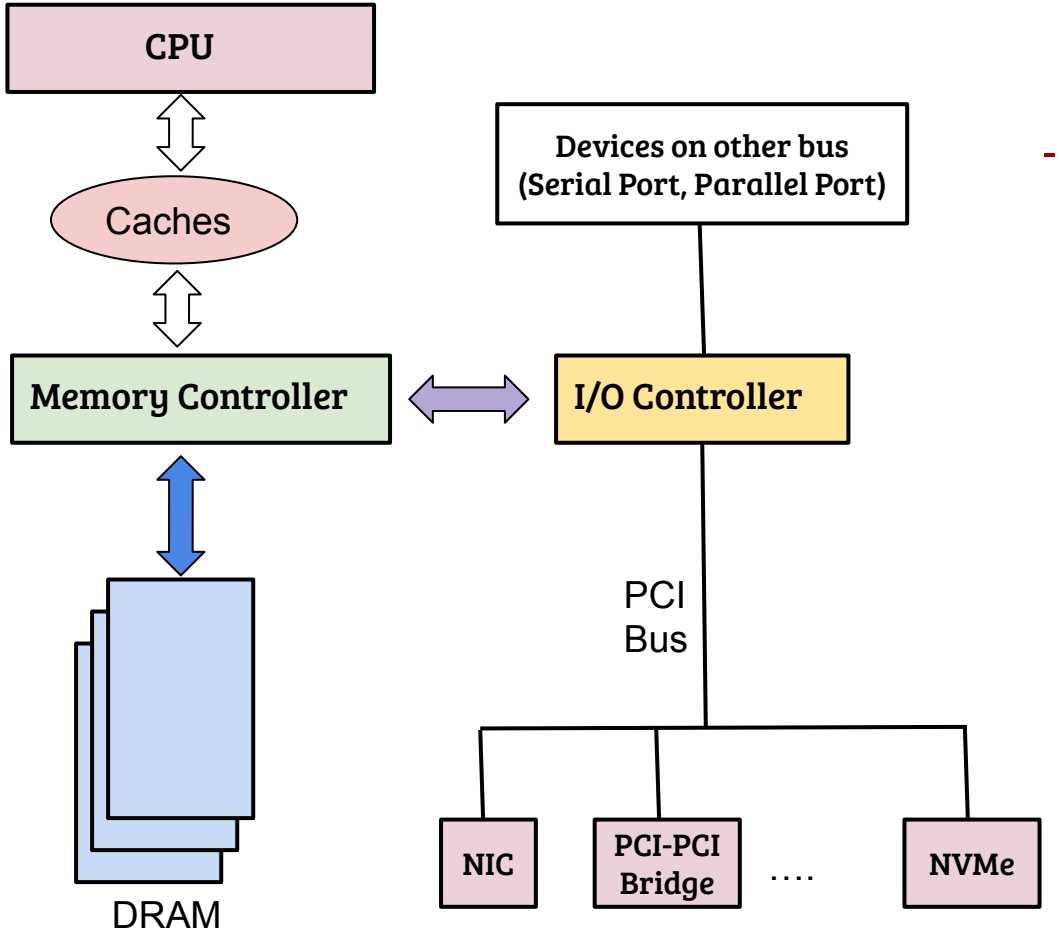
User kernel interfacing (with sysfs)

- Kernel creates virtual file systems such as procfs and sysfs
- User can operate on files hosted in sysfs using normal file system calls
- How to communicate between user and kernel using sysfs?

User kernel interfacing (with sysfs)

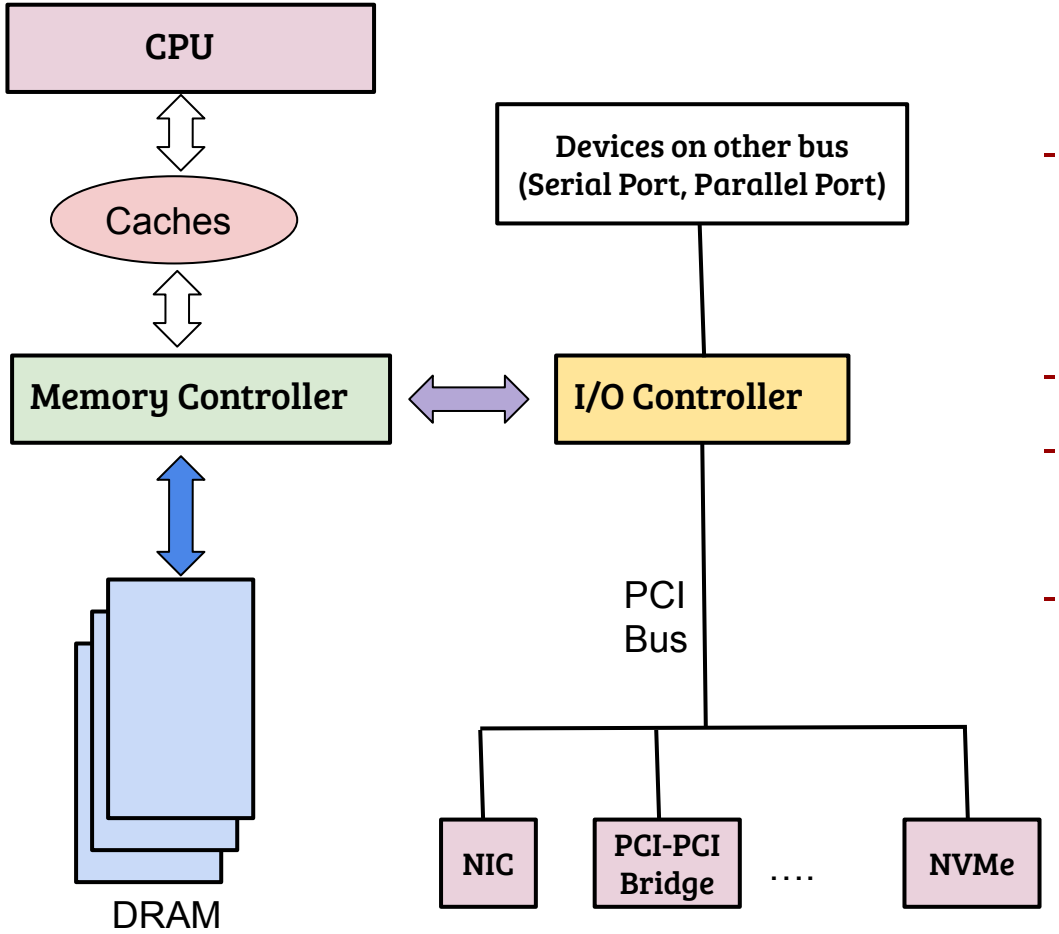
- Kernel creates virtual file systems such as procfs and sysfs
- User can operate on files hosted in sysfs using normal file system calls
- How to communicate between user and kernel using sysfs?
  - Create sysfs entries from the kernel module
  - Provide read/write handlers
  - When user performs read or write to the file, the handlers are invoked in the kernel (similar to chardev)
  - Note: Cleanup sysfs entries on module unload

I/O device interfacing (example organization)



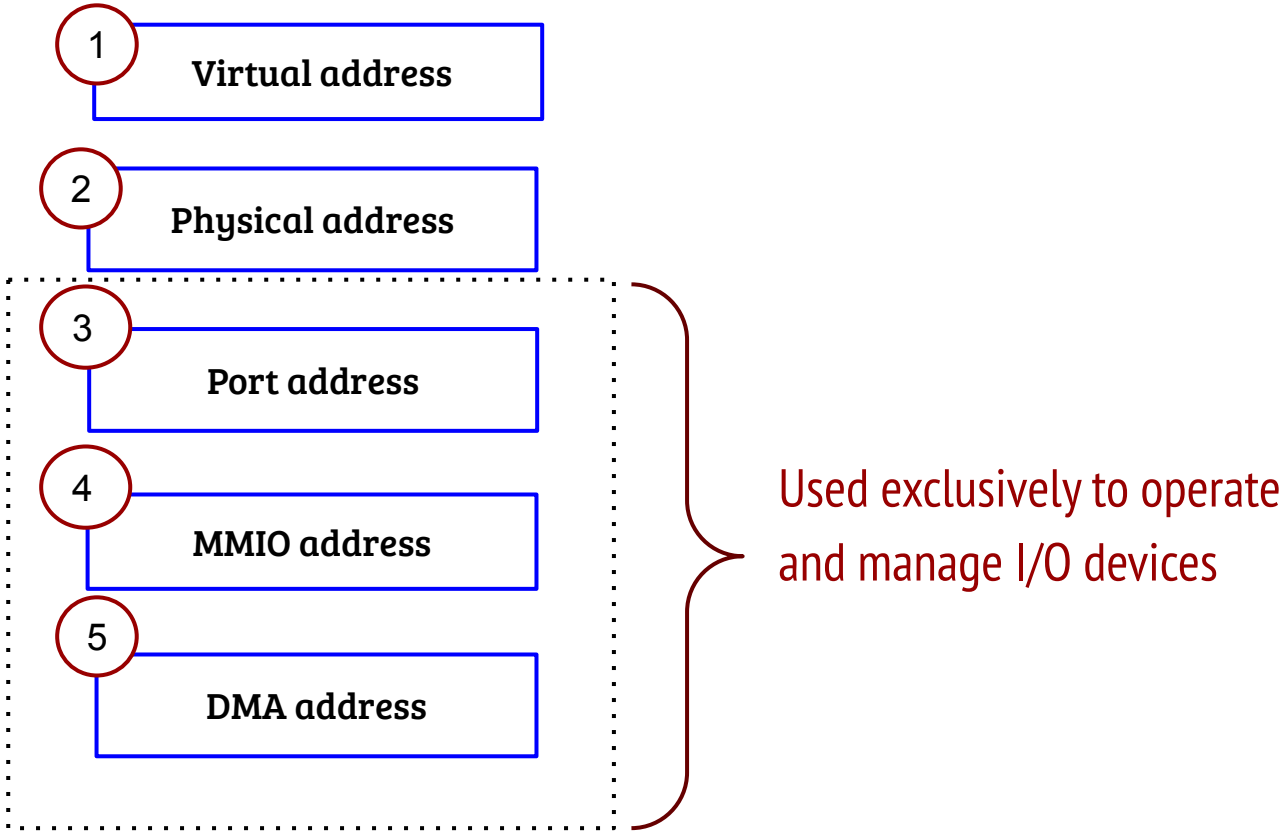
- Typically, I/O devices consists of
  - Registers (data regs (r/w), command regs, status regs etc.)
  - Memory (in-device memory, e.g., GPU memory)
  - Logic and processing (e.g, calculate a packet checksum)

I/O device interfacing (example organization)

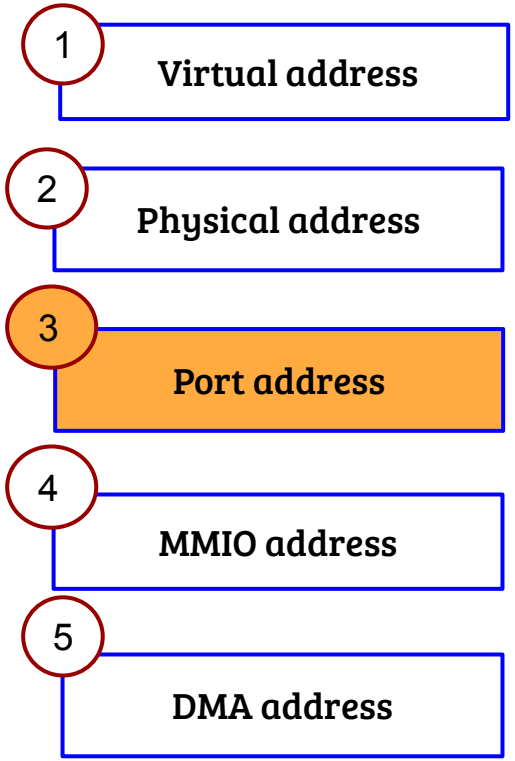


- To configure and use I/O devices, CPU should be able to operate the I/O devices (Device regs and memory)
- How to address different I/O devices?
- How to address different device resources (regs and memory)?
- Can we address the I/O devices using memory load/store instructions?

Address types in kernel



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!
- Demo: 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?
  - If the device provides a “status” port, OS can check
  - What if the device manual suggest a particular speed for an operation?

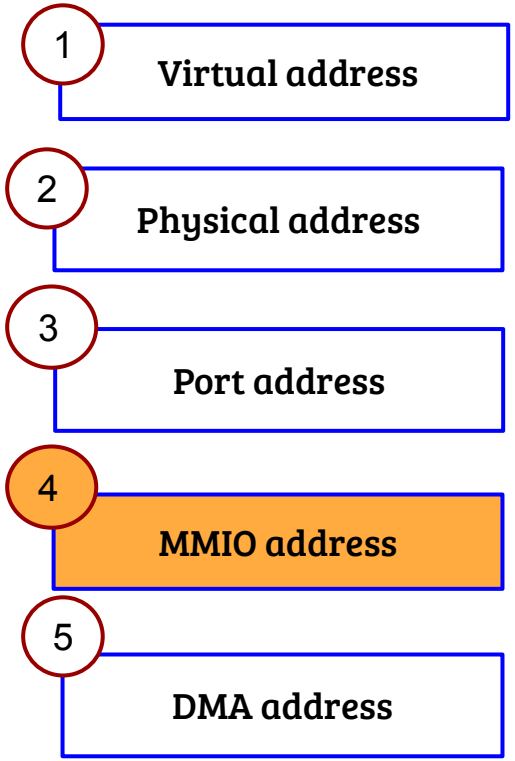
Port I/O access

- 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

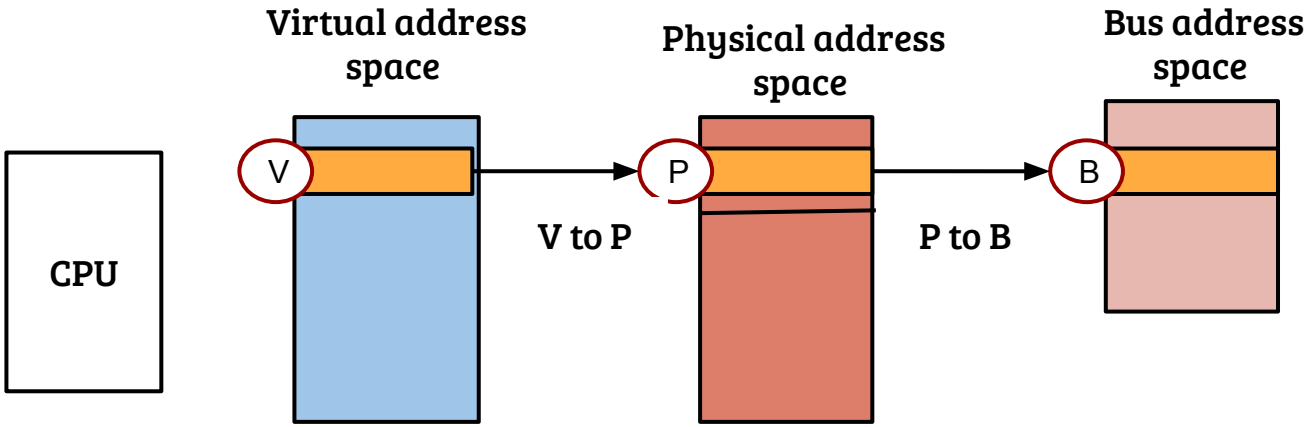
- Instructions: inb, outb, inw, outw, inl, outl
- Example: “outb $0x3F8, $0x5” ⟶ Write five to the port address 0x3F8
- Important: Completion of a write instruction may not imply the intended I/O operation is completed (CPU and I/O speeds may not match!)
- Example: To print a string into a serial console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



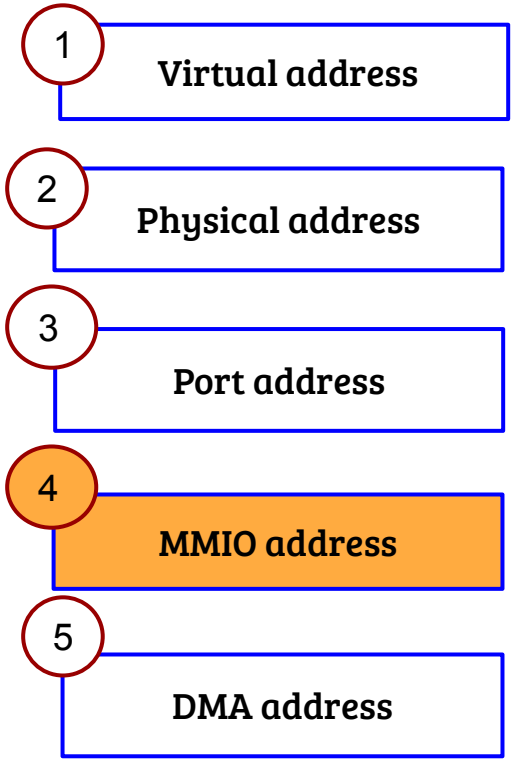
- I/O registers/memory mapped into physical address space, can be accessed like memory
- What address to use, virtual or physical?
- What extra care to be taken while accessing MMIO addresses?

Memory mapped I/O



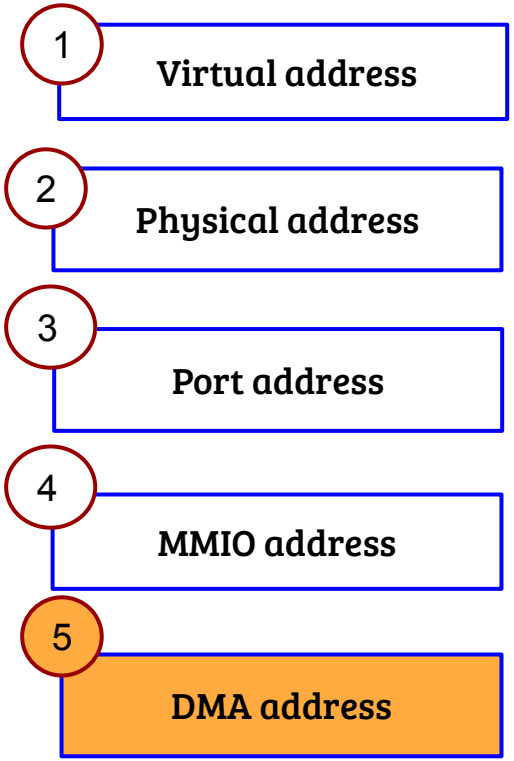
- During device discovery, kernel maintains a device to MMIO space (/proc/iomem)
- Device driver must map the PA to V before access
- Kernel source: ioremap( ), ioread32( )

Memory mapped I/O



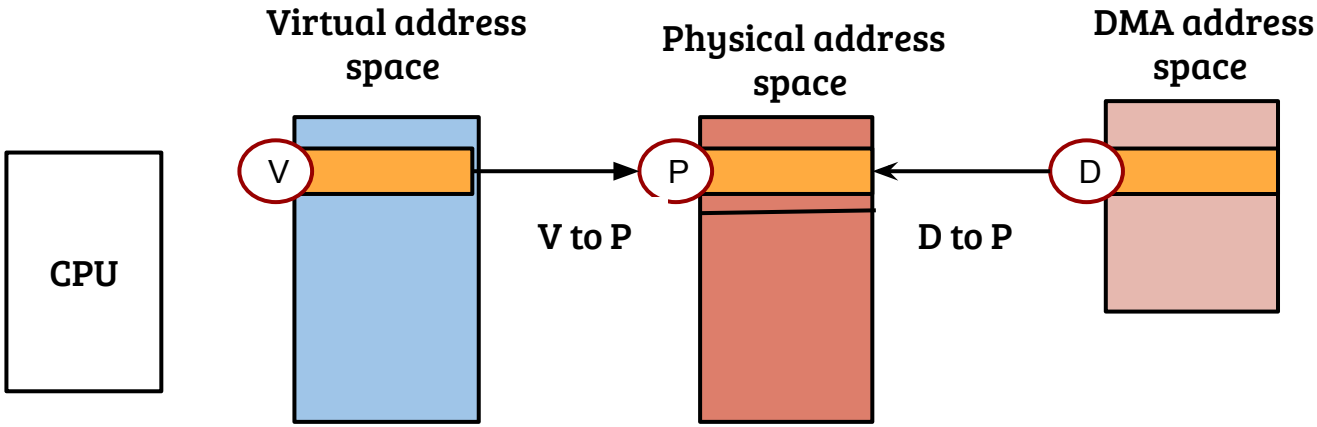
- I/O registers/memory mapped into physical address space, can be accessed like memory
- What address to use, virtual or physical?
- Virtual address
- What extra care to be taken while accessing MMIO addresses?
- Correctly timing the accesses, compiler optimizations, OOO processing

Direct memory access (DMA)



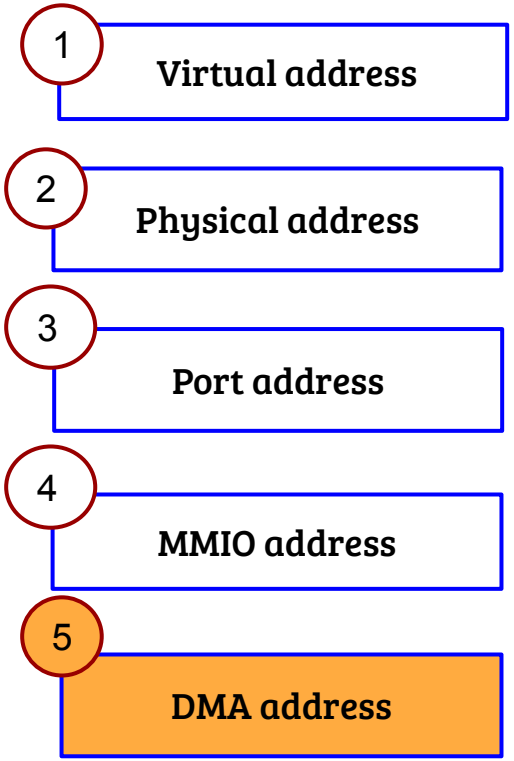
- DMA can be used if
  - DMA controller is available
  - Device supports DMA
- DMA addresses are generated and used by DMA controller
- Can be different from physical address if IOMMU is used

DMA contd.



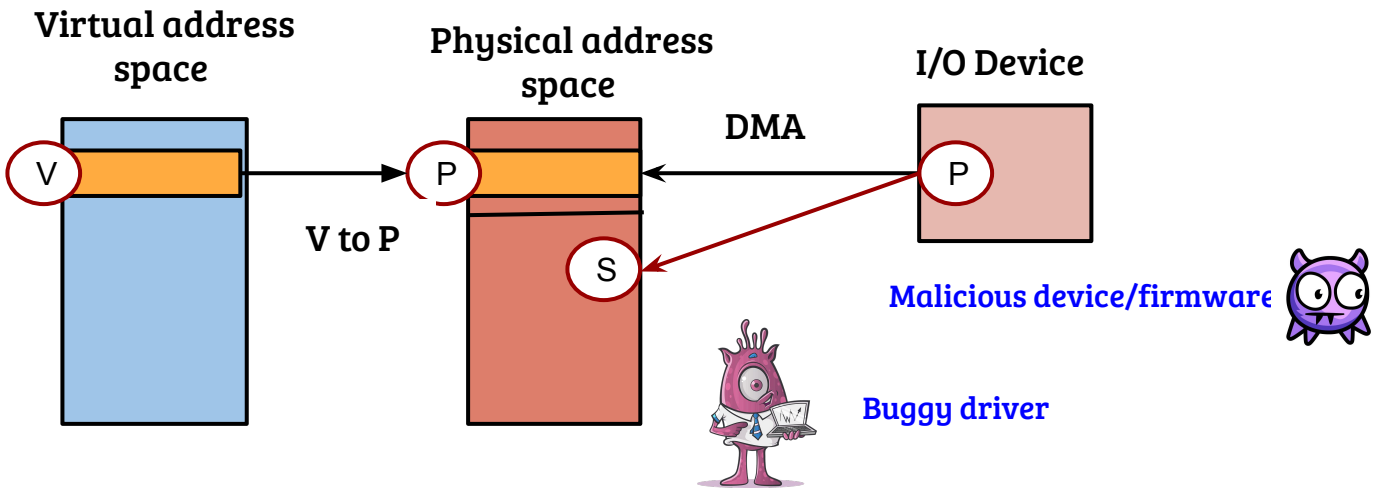
- Device driver allocates a buffer (VA = V, PA = P), no lazy allocation allowed!
- In non-IOMMU systems, device can use P directly
- With IOMMU, mapping must be setup between D ⟶ P
- The driver programmer configures the DMA address into the device (e.g., using MMIO)

Direct memory access (DMA)



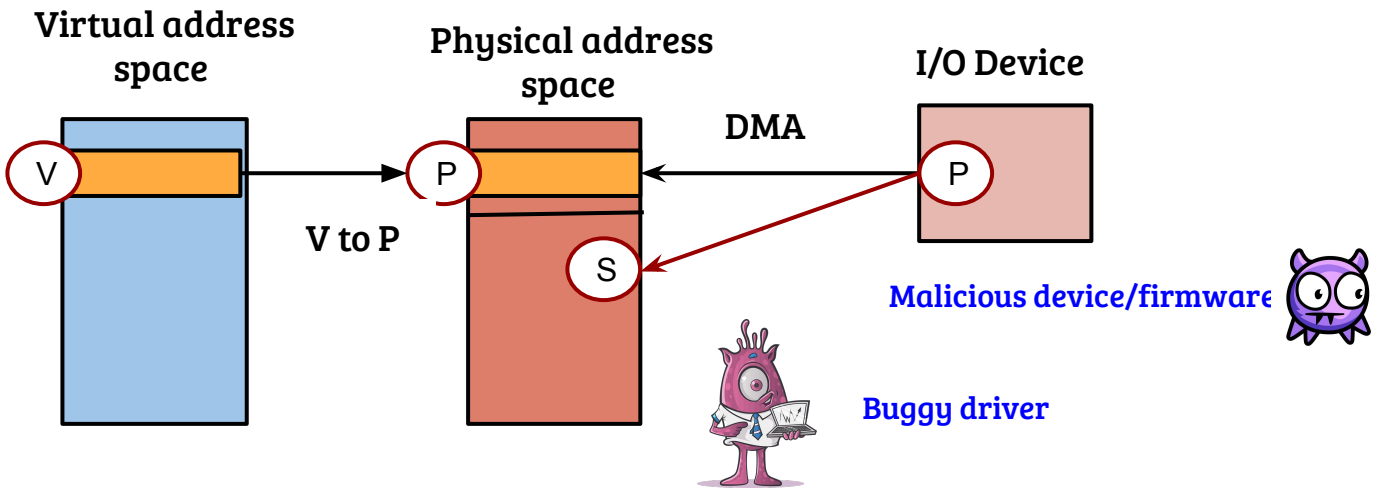
- Virtual addresses used by DMA should be mapped (don't use vmalloc( ) address)
- DMA mapping can be of two types
  - Consistent/Coherent: mostly used throughout the driver lifetime
  - Streaming/inconsistent: used to configure receive buffer of a NIC
- Refer to kernel documentation (Documentation/core-api/dma-api-howto.txt)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

Security issue with DMA



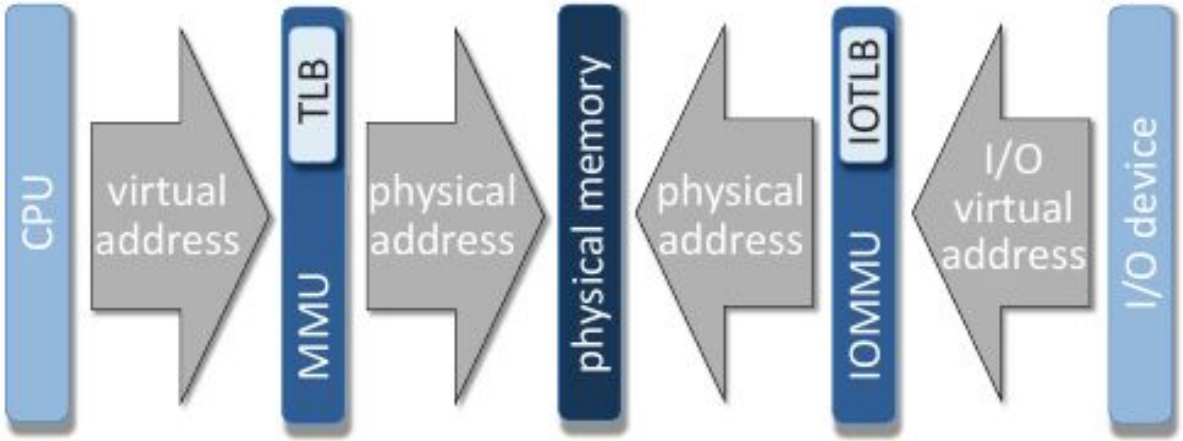
- I/O devices can access arbitrary memory locations
- Compromised security, information disclosure
- How to address this issue?

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]

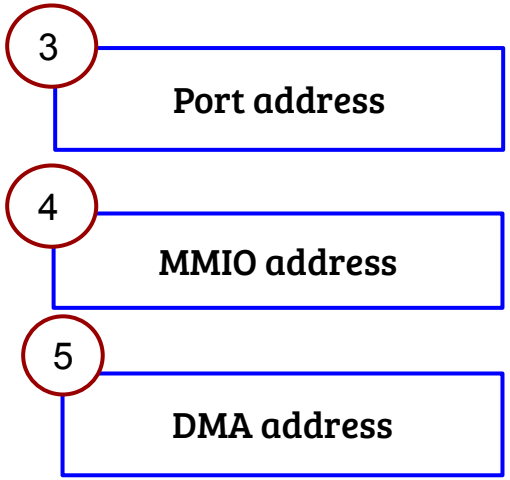


[1]

- 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!)

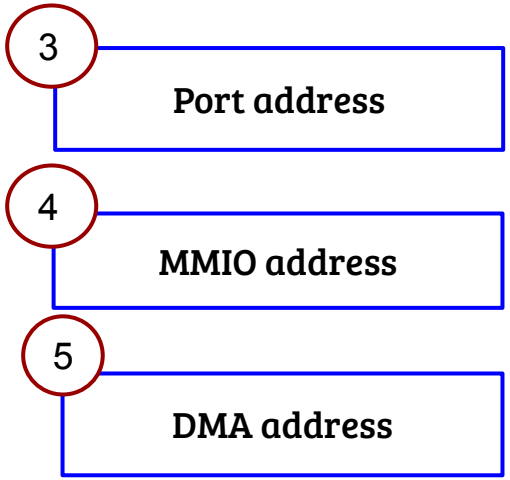
1. Malka et al. rIOMMU:Efficient IOMMU for I/O Devices that Employ Ring Buffers https://dl.acm.org/citation.cfm?id=2694355

Flexibility in I/O Addressing



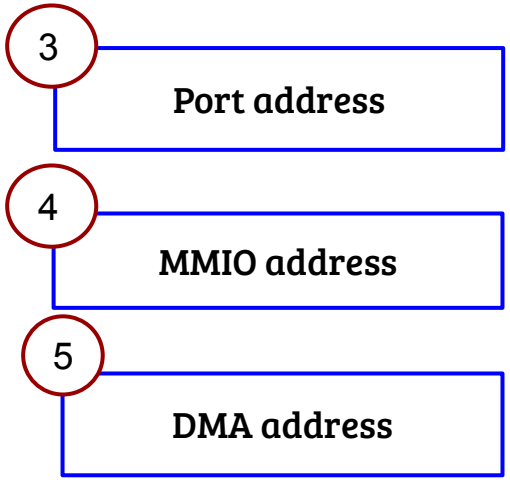
- What kind of addressing provides more flexibility to the OS, considering address as a resource?

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? Using a resource manager and mapping sub-system (e.g., PCI)