Sequence of Events

1. The kernel detects a hardware device.
2. The kernel creates an entry in sysfs (/sys).
3. The kernel generates a uevent (kernel event).
4. Udev receives this event and queries sysfs for more details.
5. Udev applies rules to name the device, set permissions, and create /dev nodes.

Sysfs (/sys) is a virtual filesystem that the Linux kernel creates to expose information about devices, drivers, and kernel subsystems. Sysfs is updated dynamically by the Linux kernel whenever devices, drivers, or other kernel objects are added or removed. This happens through kobjects, udev events, and hotplug mechanisms. Udev reads sysfs attributes and applies rules to manage /dev nodes.

Check interview question notes for smmu

|  |  |  |
| --- | --- | --- |
| Feature | User-Space Scheduling | Kernel-Space Scheduling |
| Context Switch | Full process switch (includes MMU, registers) | Lightweight, may not involve full MMU switch |
| Preemption | Can be pre-empted by a higher-priority task | Kernel threads can be preempted if preemption is enabled |
| Priority | Uses nice values and scheduler classes | Uses static kernel priorities |
| Execution Mode | Runs in user mode (ring 3) | Runs in kernel mode (ring 0) |

A user-space process can block kernel-space processes in certain conditions:

1. Blocking Due to CPU Scheduling:
2. Blocking Due to System Calls:
3. Blocking Due to Resource Contention:
4. Interrupts and Preemption:

Flow Summary (End-to-End Handoff)

1. PCIe root complex detects device (hotplug or boot).
2. Kernel PCI subsystem scans PCIe bus, reads Vendor ID, Device ID.
3. Kernel enumerates device (assigns MMIO, I/O, IRQs).
4. Kernel registers device and adds it to /sys/bus/pci/devices/.
5. Kernel matches device with driver using pci\_device\_id.
6. Kernel calls the driver’s probe function.
7. Driver initializes device (MMIO mapping, IRQ setup).
8. Device is ready for use.

Key Takeaways

* Probe is not responsible for enumeration. Enumeration happens first.
* The kernel PCI subsystem does the scanning, registering, and resource assignment.
* The probe function only initializes the device after successful matching.
* If the probe function fails, the kernel unbinds the driver and the device remains unusable.

|  |  |  |
| --- | --- | --- |
| Issue | Possible Cause | Fix |
| PASID not detected in lspci | Device doesn’t support PASID | Check `lspci -vvv |
| IOMMU translation error | Incorrect PASID or SID mapping | Verify with `dmesg |
| SR-IOV VFs not appearing | BIOS or driver issue | Enable in BIOS & load correct driver (modprobe vfio-pci) |

|  |  |  |
| --- | --- | --- |
| Step | Component Involved | Action |
| 1 | PCIe PF Driver | Creates VFs and assigns SID |
| 2 | VFIO + SR-IOV | Binds VF to vfio-pci |
| 3 | KVM + QEMU | Assigns VF to guest VM |
| 4 | PCIe Root Complex | Tags transactions with PASID |
| 5 | IOMMU (SMMU/VT-d) | Maps PASID to guest memory |
| 6 | PCIe Device (VF) | Completes DMA with PASID isolation |

|  |  |  |
| --- | --- | --- |
| Step | Component | Action |
| 1 | Guest VM | Issues DMA request with PASID |
| 2 | PCIe Root Complex | Tags transaction with PASID & SID |
| 3 | SMMU (IOMMU) | Looks up PASID table in SMMU\_PMCFGn |
| 4 | SMMU | Translates guest virtual address to host physical address |
| 5 | PCIe Device (VF) | Accesses translated memory |

|  |  |  |
| --- | --- | --- |
| Field | Bits | Description |
| Type | 5 | Defines request type (e.g., Memory Read, Memory Write) |
| TC (Traffic Class) | 3 | Priority classification |
| Attributes | 2 | No Snoop, Relaxed Ordering |
| Length | 10 | Number of DWORDs |
| Requester ID | 16 | Bus/Device/Function number |
| Tag | 8 | Identifies unique transactions |
| Address | 64 | Target memory address |
| PASID (Optional) | 20 | Identifies process address space |

|  |  |
| --- | --- |
| Step | Action |
| 1 | VF sends TLP with PASID |
| 2 | Root Complex forwards to SMMU |
| 3 | SMMU translates PASID to Physical Address |
| 4 | Translated request sent to memory controller |
| 5 | Completion TLP returned to VF |

|  |  |  |
| --- | --- | --- |
| Step | Component | Action |
| 1 | PASID Table (SMMU\_PMCFGn) | Maps PASID to Context Descriptor |
| 2 | PASID Context Descriptor | Stores TTBR0, TTBR1 for page table lookup |
| 3 | Page Table Walk | L1 → L2 → Final Physical Address |
| 4 | Translated Address | Returned to PCIe device |

|  |  |  |
| --- | --- | --- |
| TLP Field | Hexadecimal Value | Description |
| TLP Type | 0x4A | Memory Write with PASID |
| Requestor ID | 0x0002 | PCIe Device ID (GPU/NIC) |
| PASID | 0x00000003 | Process Address Space ID (TensorFlow) |
| ATS Flag | 0x1 | Address Translation Service Enabled |
| Virtual Address | 0x30000000 | Target Virtual Address |
| Length | 0x1000 | Size of data in bytes (4KB) |
| Data Payload | 0xDEADBEEF | Data being transferred |

The device constructs a TLP (Transaction Layer Packet) like this:

|  |  |
| --- | --- |
| Field | Value |
| TLP Type | Memory Write Request |
| Request ID | Device BDF (Bus, Device, Function) |
| Address | Physical Memory Address in RAM |
| Data | Actual Payload (Data to write) |

|  |  |  |  |
| --- | --- | --- | --- |
| Interrupt Type | Handled by | Routing Mechanism | Use Case |
| SGI (Software Generated Interrupts) | Redistributor (GICR) | Sent by software to specific cores | Inter-core communication |
| PPI (Private Peripheral Interrupts) | Redistributor (GICR) | Always local to the core | Timers, watchdogs |
| SPI (Shared Peripheral Interrupts) | Distributor (GICD) | Routed to multiple cores | Shared peripherals like GPU, PCIe |
| LPI (Locality-specific Peripheral Interrupts) | Redistributor (GICR) + ITS | Message-based (memory-mapped) | Power-efficient interrupts |

The GIC consists of the following components:

1. Distributor (GICD)
   * Manages global interrupts (SPI, LPI, PPI). Routes interrupts to appropriate CPU interfaces. Controls enable/disable and priority settings.
2. Redistributor (GICR) [Introduced in GICv3]
   * Manages per-core interrupts (PPI, LPI). Each core has a separate redistributor. Essential for handling power-efficient LPIs.

|  |  |  |
| --- | --- | --- |
| Step | Component | Register Used |
| Enable GIC | Distributor (GICD) | GICD\_CTLR |
| Configure SPI | Distributor (GICD) | GICD\_IPRIORITYR, GICD\_ICFGR |
| Enable Interrupt | Distributor (GICD) | GICD\_ISENABLER |
| Enable CPU Interface | CPU Interface (ICC) | ICC\_CTLR, ICC\_PMR |
| Read IRQ ID | CPU Interface (ICC) | ICC\_IAR |
| Handle Interrupt | Software ISR | - |
| Complete Interrupt | CPU Interface (ICC) | ICC\_EOIR |

|  |  |  |
| --- | --- | --- |
| Step | Action | Component |
| Enable Redistributor | Wake up each CPU's Redistributor | GICR |
| Set LPI Configuration | Define priorities in memory | PROPBASER |
| Configure ITS | Map device events to LPIs | ITS |
| Enable LPIs | Activate LPIs in Redistributor | GICR |
| Route SPI to Multiple CPUs | Target multiple CPUs | GICD |
| Handle LPI in ISR | Read LPI ID and clear | CPU Interface |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Driver Model | Discovery | | Use Case | | Example |
| Bus Driver | Self-discoverable | | Handles devices on dynamic buses. Best for devices that can self-identify and are connected via a standard bus. If the device is attached to a discoverable bus, the Linux kernel already has a subsystem to manage it. | | PCIe, USB, I2C, SPI |
| Platform Driver | Non-discoverable | | Manages fixed hardware components. If the device is part of the fixed hardware and needs manual registration in the device tree. | | SoC components, PCIe root complex, GPIO controllers, UART, power controller |
| Character Driver | Non-discoverable | | Provides user-space interaction for character devices. Used when a device provides byte-stream data access. | | Serial ports (/dev/ttyS0), sensors, touchscreen |
| Block Driver | Non-discoverable | | Manages block storage devices. If the device requires block-based read/write operations. | | HDDs, SSDs (/dev/sda), eMMC, SD card |
| Network Driver | Either | | Handles network interfaces | | Ethernet, Wi-Fi (struct net\_device) |
| Virtual Driver | Depends | | Manages virtualized hardware | | Virtio, Hypervisor drivers, Hyper-V synthetic drivers, Docker, VM network drivers |
| Driver Model | | Uses Device Tree? | | Why? | |
| Platform Driver | | ✅ Yes | | Needed to describe fixed hardware in SoCs (e.g., PCIe root complex, GPIOs, clocks). | |
| Bus Driver (PCIe, I2C, SPI) | | ✅ Yes (sometimes) | | Used when the bus itself needs configuration (e.g., PCIe controllers in SoCs). | |
| Character Driver | | ❌ No (mostly) | | Devices like serial ports often don't need DT, but some embedded devices do (e.g., sensors over I2C). | |
| Block Driver | | ❌ No | | Storage devices (eMMC, SD) are usually detected via standard mechanisms. | |
| Network Driver | | ✅ Yes (for SoC NICs) | | SoC Ethernet controllers need DT, but PCI/USB NICs don’t. | |
| Virtual Driver | | ❌ No | | Virtual devices (e.g., Virtio, Hyper-V) don’t need DT as they are software-based. | |

|  |  |  |
| --- | --- | --- |
| Driver Model | Initialization Function | When It Gets Called? |
| Platform Driver | probe(struct platform\_device \*pdev) | When a device matching the Device Tree (DT) or ACPI table is found. |
| Bus Driver (PCI, USB, I2C, SPI) | probe(struct pci\_device \*pdev), probe(struct usb\_interface \*intf), probe(struct i2c\_client \*client) | When the device is detected on the respective bus. |
| Character Driver | open(struct inode \*inode, struct file \*filp) | When a user-space application opens the device file (/dev/mydevice). |
| Block Driver | request\_queue\_fn(struct request\_queue \*q, struct request \*req) | When the OS issues a read/write request to the block device. |
| Network Driver | ndo\_open(struct net\_device \*dev) | When the network interface is brought up (ifconfig eth0 up). |
| Virtual Driver | probe(struct virtio\_device \*vdev) | When the virtual device is initialized by the hypervisor. |

* For PCIe device drivers, entry points are:
  + module\_init() → pci\_register\_driver() → probe()
* For PCIe host controllers, entry points are:
  + platform\_driver\_register() → pci\_host\_probe()

|  |  |
| --- | --- |
| Concept | Must-Know Details |
| Bounce Buffer | Prevents DMA failures in low-memory situations. |
| Swiotlb (Software IOMMU) | Handles large DMA transfers without IOMMU. |
| Scatter-Gather DMA | Enables fragmented memory DMA. |
| IOMMU Bypass Mode | Direct DMA without IOMMU translation. |
| Coherent vs Streaming DMA | Coherent = cache consistent; Streaming = faster. |

Advantages of Coherent DMA

|  |  |
| --- | --- |
| Feature | Benefit |
| Cache coherence | No need to flush/invalidate cache manually. |
| Simple API | Just allocate memory using dma\_alloc\_coherent(). |
| Predictable behavior | Data written by device is instantly visible to CPU. |

Disadvantages of Coherent DMA

|  |  |
| --- | --- |
| Limitation | Problem |
| Non-cached memory | Since cache is bypassed, performance is low. |
| Memory consumption | Requires physically contiguous memory. |
| Low throughput | Not suitable for high-speed data transfers. |

Use Cases for Coherent DMA

|  |  |
| --- | --- |
| Scenario | Example |
| Low data throughput devices | I2C, UART, SPI, simple sensors. |
| Memory-mapped devices | Framebuffer for small embedded displays. |
| Control Data Buffers | Device descriptors, small control messages. |

Advantages of Streaming DMA

|  |  |
| --- | --- |
| Feature | Benefit |
| High throughput | Uses cache, ensuring faster data transfer. |
| Dynamic allocation | Can use existing kernel buffers. |
| Efficient for bulk data | Best for PCIe, networking, audio, etc. |

Disadvantages of Streaming DMA

|  |  |
| --- | --- |
| Limitation | Problem |
| Cache management | Manual cache flush/invalidate required. |
| Data inconsistency | Device may read stale data if cache not flushed. |
| Complex API | Requires manual map/unmap. |

Use Cases for Streaming DMA

|  |  |
| --- | --- |
| Scenario | Example |
| High-speed data transfer | PCIe, NVMe, network adapters. |
| Audio/Video Streaming | HDMI, camera input, sound cards. |
| Storage Devices | NVMe, SSD, hard drives. |

|  |  |  |
| --- | --- | --- |
| Feature | Coherent DMA | Streaming DMA |
| Memory Type | Non-cached (Uncached). | Cached (High-speed memory). |
| Cache Coherency | Always coherent. No flush/invalidate needed. | Needs manual cache flush/invalidate. |
| Performance | Slower (since cache is bypassed). | Higher throughput (cache utilized). |
| API Simplicity | Simple (dma\_alloc\_coherent()). | Complex (dma\_map\_single()). |
| Use Case | Low-speed devices, control buffers. | High-speed data streams, PCIe, network. |
| CPU Involvement | Minimal. CPU doesn't cache. | Higher CPU involvement due to cache management. |
| Memory Requirement | Requires physically contiguous memory. | Uses virtually mapped pages. |
| Throughput | Low throughput. | High throughput. |

|  |  |
| --- | --- |
| Scenario | Recommended DMA |
| Small control messages, descriptors, command buffers. | ✅ Coherent DMA. |
| High-speed bulk data transfer (Network, Video, PCIe). | ✅ Streaming DMA. |
| UART, I2C, SPI control data. | ✅ Coherent DMA. |
| NVMe SSD, Network Adapter, Camera Feed. | ✅ Streaming DMA. |

|  |  |  |
| --- | --- | --- |
| Feature | Coherent DMA | Streaming DMA |
| Memory Type | Non-cached. | Cached (faster). |
| Cache Management | Automatic (no flush needed). | Manual (flush needed). |
| Use Case | Control buffers, descriptors. | High-speed data transfer. |
| Performance | Lower throughput. | High throughput. |
| API | Simple (dma\_alloc\_coherent). | Complex (dma\_map\_single). |

Complete PCIe DMA Flow (Diagram)

|  |  |  |  |
| --- | --- | --- | --- |
| Step | PCIe Device Action | PCIe Root Complex (RC) Action | Result |
| 1 | Send Memory Write Request (TLP). | Accept TLP. | Data sent. |
| 2 | Wait for Completion TLP. | Write to memory. | Data written. |
| 3 | Send Memory Read Request. | Fetch data from DRAM. | Data returned. |
| 4 | Receive Completion TLP. | Done. | Data received. |

PCIe DMA + IOMMU Key Takeaways

|  |  |  |
| --- | --- | --- |
| Feature | Without IOMMU | With IOMMU |
| Memory Safety | No protection | Protected |
| Mapping | Physical memory | IOMMU mapped |
| Crash Risk | High | Zero |
| Virtualization | Impossible | Fully isolated |

Who Controls Which?

|  |  |  |
| --- | --- | --- |
| Memory Type | Who Controls Access? | Who Does Translation? |
| Host Memory (RAM) | Root Complex (RC) | IOMMU (in RC) |
| Device Internal Memory | PCIe Device itself (like BlueField) | SMMU (inside the Device) |
| Remote PCIe Memory | Root Complex | IOMMU (if mapped) |

|  |  |  |
| --- | --- | --- |
| Feature | Character Device Driver | Block Device Driver |
| Data Handling | Stream of bytes | Blocks of data |
| Example | Serial ports, Keyboards | Hard disks, SSDs |
| Buffering | No automatic buffering | Uses caching |
| System Calls | read(), write() | read(), write(), fsync() |

What is the difference between spinlock() and mutex()?

|  |  |  |
| --- | --- | --- |
| Feature | Spinlock | Mutex |
| Blocking | No (Busy-wait) | Yes |
| Performance | Fast in SMP | Slower due to scheduling |
| Usage | Short critical sections | Long critical sections |

|  |  |
| --- | --- |
| Feature | API |
| Register device | alloc\_chrdev\_region(), cdev\_add() |
| File operations | open(), read(), write(), release() |
| IOCTL support | unlocked\_ioctl() |
| Memory mapping | mmap() |
| Interrupts | request\_irq(), free\_irq() |
| Device cleanup | cdev\_del(), unregister\_chrdev\_region() |

|  |  |
| --- | --- |
| Function Call | Purpose |
| iommu\_map() | Maps device virtual address to physical memory. |
| iommu\_unmap() | Unmaps a device memory range. |
| iommu\_attach\_device() | Attaches a device to an SMMU domain. |
| iommu\_create\_domain() | Creates a new SMMU translation domain. |

What You Should Master to Crack Interview

|  |  |
| --- | --- |
| Area | Deep Knowledge Required |
| Kernel Driver Flow | Understand SMMU driver initialization. |
| DMA Mapping | Explain iommu\_map(), dma\_map\_single(). |
| Virtualization Flow | Understand VFIO, KVM, and SMMU interaction. |
| Page Tables | Explain L1, L2, L3 translation. |
| Fault Handling | Explain Permission Fault, Translation Fault. |

|  |  |
| --- | --- |
| Feature | Function |
| ATS (Address Translation Service) | Allows PCIe devices to cache translations. |
| PRI (Page Request Interface) | Allows PCIe devices to request new pages. |
| PASID (Process Address Space ID) | Allows multiple processes to share one device. |

|  |  |
| --- | --- |
| Feature | Purpose |
| ATS | Device caches translation |
| PRI | Device requests page faults |
| PASID | Allows multi-process DMA |

|  |  |  |
| --- | --- | --- |
| Step | Action | Result |
| 1 | PCIe device calls dma\_map\_single() | Kernel maps memory. |
| 2 | Kernel calls iommu\_map() | SMMU maps Virtual Address to Physical. |
| 3 | SMMU updates page tables | Address translation established. |
| 4 | Device can now do DMA | Successful memory access. |

|  |  |
| --- | --- |
| Question | Expected Answer |
| How does PCIe device do DMA? | Explain Stream ID → Context Bank → Page Table. |
| How does VM PCIe Passthrough work? | Explain Two-Stage Translation (GVA→GPA→HPA). |
| What happens on Page Fault? | Explain PRI, SMMU IRQ, and Translation Fault. |
| How does ATS improve performance? | Device caches Address Translations. |

|  |  |
| --- | --- |
| Feature | Benefit |
| PASID | Allows multi-process DMA on one device. |
| SR-IOV | Allows one PCIe device to serve multiple VMs. |
| SMMU | Ensures full memory isolation. |
| ATS | Allows device-side translation caching. |
| PRI | Handles page faults during DMA. |

|  |  |
| --- | --- |
| Question | Expected Answer |
| What is PASID? | Unique ID per process for isolated DMA. |
| How does SMMU handle PASID? | Maps PASID to separate page tables. |
| What is SR-IOV? | One PCIe device emulating multiple VFs. |
| How does PASID + SR-IOV work? | PASID isolates process memory within VFs. |
| How does PCIe device request memory? | Stream ID + PASID → Page Table Walk. |

|  |  |  |
| --- | --- | --- |
| Step | Action | Result |
| 1 | PCIe device wants DMA | Sends PASID + VA |
| 2 | Device caches translation (ATS Cache) | Fast memory access |
| 3 | Cache Miss → Sends Page Request (PRM) | Asks SMMU for translation |
| 4 | SMMU performs page walk | Returns Physical Address |
| 5 | Device updates ATS Cache | Reduces future latency |
| 6 | Future DMA (Cache Hit) | Avoids SMMU entirely |
| 7 | Page Eviction (PRI Trigger) | Handles memory swapping dynamically |

Why ATS Is a Game-Changer?

|  |  |  |
| --- | --- | --- |
| Feature | Without ATS | With ATS |
| Latency | High (SMMU page walks) | Near-zero (Cache hits) |
| Throughput | Low (Blocking) | High (Concurrent DMA) |
| Page Fault | Kernel handles it | Device sends PRI |
| PASID Support | Single process | Multi-process support |

|  |  |
| --- | --- |
| Question | Expected Answer |
| What is ATS? | Device-side address translation caching. |
| Why does PCIe need ATS? | To reduce SMMU page walks and increase performance. |
| What happens on Cache Miss? | Device sends a Page Request (PR) to SMMU. |
| What is PRI? | Page Request Interface for handling page faults. |
| How does ATS + PASID work together? | PASID isolates processes, ATS caches translations. |

|  |  |
| --- | --- |
| Action | Kernel Flow |
| DMA Map | dma\_map\_single() → iommu\_map() → arm\_smmu\_map() |
| Page Request | arm\_smmu\_handle\_pri() → arm\_smmu\_map() |
| Cache Hit | Device Bypasses SMMU Using ATS Cache |
| Page Fault | PRI Fault → Page Response → Resume DMA |

|  |  |  |
| --- | --- | --- |
| Step | Flow | Responsible Code |
| 1 | ioctl(fd, cmd, arg) | User-space application |
| 2 | sys\_ioctl() | Kernel system call |
| 3 | file->f\_op->unlocked\_ioctl() | File operations |
| 4 | my\_ioctl() in driver | Driver-specific logic |
| 5 | copy\_from\_user() | Move data from user to kernel |
| 6 | copy\_to\_user() | Move data from kernel to user |
| 7 | close(fd) | Trigger release() in driver |

GPU kernel interview questions:

|  |  |
| --- | --- |
| Component | Description |
| Distributor (GICD) | Manages interrupt distribution to CPU interfaces. |
| Redistributor (GICR) | Manages interrupts for individual CPUs and power management. |
| CPU Interface (GIC-CPU) | Delivers interrupts to CPU cores. |
| ITS (Interrupt Translation Service) | Handles Message Signalled Interrupts (MSI) from PCIe devices. |

Interrupt Types in GICv3

1. SPIs (Shared Peripheral Interrupts): For devices shared across multiple CPUs (like PCIe devices).
2. PPIs (Private Peripheral Interrupts): Private to each CPU core (like local timers).
3. SGIs (Software Generated Interrupts): Raised by software for inter-core communication.
4. LPIs (Locality-specific Peripheral Interrupts): Delivered using ITS for high-performance devices.

|  |  |
| --- | --- |
| Feature | Purpose |
| Virtual CPU Interface (vGIC) | Allows Guest VMs to receive interrupts. |
| List Registers (LR) | Holds virtual interrupts for Guest VMs. |
| VM Exit on IRQ | Exits Guest mode when IRQ occurs. |
| Direct Injection | Allows hypervisor to inject virtual interrupts. |

Here’s the complete call stack from hardware IRQ to driver:

Hardware Device

↓

GICv3 Distributor

↓

GICv3 Redistributor

↓

CPU Interface (IRQ asserted)

↓

entry.S (vector table)

↓

do\_IRQ()

↓

handle\_IRQ()

↓

generic\_handle\_domain\_irq()

↓

irq\_desc->handle\_irq()

↓

device\_irq\_handler()

↓

write\_icc\_eoir1\_el1()

↓

return\_from\_irq()

Handling MSI with ITS (Interrupt Translation Service): If your device supports Message Signaled Interrupts (MSI) like PCIe, the flow changes:

1. Device sends MSI → GICv3 ITS receives MSI.
2. ITS translates MSI to LPI (Locality Specific Interrupt).
3. GICv3 Redistributor receives LPI.
4. Kernel flow remains the same.

do\_IRQ()

→ handle\_IRQ()

→ irq\_desc->handle\_irq()

→ PCIe device driver

What Does ITS (Interrupt Translation Service) Do?

* ITS handles MSI (Message Signaled Interrupts) from PCIe devices.
* It translates MSI → LPI (Locality-specific Peripheral Interrupt).
* GICR then delivers the LPI to the target CPU.

LPI Flow After Translation

PCIe Device → MSI → ITS → LPI → GICR → CPU

Kernel Flow for ITS

1. Kernel initializes ITS driver:

drivers/irqchip/irq-gic-v3-its.c

1. Maps MSI to LPI:

gic\_msi\_init(dev, irq);

1. Sends EOI:

write\_icc\_eoir1\_el1(irq);

1. fork()

* The new process is an exact copy of the parent. It has its own memory space, file descriptors, and process ID with COW.
* Returns 0 in the child and child PID in the parent. Both parent and child continue execution from the same point.

2. exec()

* Replaces the current process with a new program. It does not return on success (the old process is overwritten). After fork(), the child process usually calls exec() to load and run a different program. The old process image (code, data, stack) is completely replaced. Used after fork() to execute a different program in the child. S, chid can execute without affecting parent. But if directly exec is called in parent without calling fork() then parent process gets replaced and dies.

Common Variants of exec:

* execl(), execv(), execle(), execve(), execlp(), execvp()

3. clone()

* More flexible than fork(), used for creating threads or lightweight processes. Allows sharing specific resources between parent and child (e.g., memory, file descriptors).
* Used internally by pthread\_create() for thread creation. Requires a stack for the child process.

|  |  |  |
| --- | --- | --- |
| Function | Parameter | Description |
| copy\_to\_user() | to | Pointer to user space buffer (\_\_user indicates user space) |
|  | from | Pointer to kernel space buffer |
|  | n | Number of bytes to copy |
| copy\_from\_user() | to | Pointer to kernel space buffer |
|  | from | Pointer to user space buffer (\_\_user indicates user space) |
|  | n | Number of bytes to copy |

copy\_from\_user()

├── access\_ok()

├── \_\_copy\_from\_user()

│ ├── raw\_copy\_from\_user()

│ │ ├── Page fault may occur here

Page Fault

├── handle\_mm\_fault()

│ ├── do\_page\_fault()

│ │ ├── handle\_pte\_fault()

│ │ ├── bring page from disk (if swapped out)

│ │ ├── create mapping in MMU

│ │ ├── retry copy operation

Why Is DMA With User Space Complicated? In normal DMA: Device → Kernel Memory → User Space. But if you directly map user space memory to DMA, it eliminates an extra copy.

Kernel Mapping User Space for DMA: The kernel must pin user space memory to:

* Prevent it from being swapped.
* Translate virtual address → physical address.
* DMA Unmap: Once the transfer is done:
* dma\_unmap\_single(dev, dma\_addr, PAGE\_SIZE, DMA\_TO\_DEVICE);
* put\_page(pages[i]);
* Unpins the pages. Flushes cache if required.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Method | CPU Copy? | Cache Invalidation? | Swapping Risk? | Speed |
| copy\_from\_user() | Yes | No | Yes | Slow |
| DMA with user space | No (device does it) | Yes (if required) | No (pinned memory) | Blazing Fast |

DMA Is Used In:

|  |  |
| --- | --- |
| Device | Data Transfer |
| Network Card | Huge data from NIC → User Space |
| GPU | Framebuffer directly to user space |
| SSD | Disk → User Space (Zero Copy) |

Potential Problems With User Space DMA

|  |  |
| --- | --- |
| Problem | Why? |
| Security Risk | Device may read user space secrets. |
| Page Faults | If memory is swapped, transfer fails. |
| Cache Incoherency | If cache isn't flushed, stale data may appear. |

Cache Types in Linux: The two important caches in Linux:

1. Write-back Cache (WB):
   * Most common.
   * Modified data stays in cache → not immediately written to RAM.
2. Write-through Cache (WT):
   * Writes go directly to RAM.
   * Less performance, higher consistency.

|  |  |  |
| --- | --- | --- |
| API | What It Does | When To Use |
| dma\_sync\_single\_for\_device() | Flush cache to memory | Before device reads memory |
| dma\_sync\_single\_for\_cpu() | Invalidate cache | After device writes memory |

How Does DPDK Avoid Kernel Copy? Instead of using:

copy\_from\_user()

DPDK:

* Directly maps user space memory to DMA buffer.
* Completely bypasses the kernel.

Step 1: Memory Pinning

DPDK first pins user memory using:

get\_user\_pages()

This prevents: Page swapping and Page migration.

Step 2: DMA Mapping: Then DPDK maps user space memory for DMA:

dma\_map\_single();

Which translates:

User Space Virtual Address → Physical Address

This allows NIC to directly write to user space.

Step 3: Device Writes to User Space Buffer: The device now:

* Directly uses DMA.
* Avoids kernel copy.

Step 4: Cache Invalidation

After DMA, the user space cache is stale. So DPDK does:

dma\_sync\_single\_for\_cpu();

Which forces the cache to:

* Fetch from RAM (not cache).
* Avoid stale data.

Result: Insane Performance

|  |  |  |
| --- | --- | --- |
| Operation | Traditional Networking | DPDK Zero-Copy |
| Kernel Copy | Yes | No |
| Interrupt Overhead | High | Zero (polling mode) |
| Throughput | ~10 Gbps | 100+ Gbps |

CMA efficiently allocates large contiguous physical memory for devices like GPUs, multimedia codecs, cameras, and other hardware that require physically contiguous memory. Addresses the limitations of traditional page-based allocators (like kmalloc/get\_free\_pages) that may struggle to find large contiguous blocks after prolonged system uptime.

Direct Memory Access devices often require large physically contiguous memory to efficiently transfer data between devices without CPU involvement.

CMA uses a pre-reserved memory region in physical memory that is contiguous. This region is carved out at boot time and kept reserved for devices that require contiguous memory.

When a device driver needs contiguous memory, it calls APIs like: dma\_alloc\_coherent(); Internally, dma\_alloc\_coherent() calls CMA functions. CMA looks into its reserved memory pool for a large enough contiguous block. If found, it returns the physical memory address.

If CMA runs out of reserved memory, it can allocate from normal page memory using the migrate mechanism. It will then migrate movable pages (like user-space memory) to create a contiguous block. Once memory is allocated, it is pinned and cannot be reclaimed until freed.

Page migration allows CMA to move movable pages (user-space memory, file caches, etc.) to make space for contiguous allocations. Pages marked as movable can be evicted to different physical locations. This increases the probability of large contiguous memory availability.

If normal memory is exhausted and CMA has unused reserved memory, the kernel can temporarily utilize CMA’s memory pool. This ensures that memory never goes to waste even if the device doesn’t use it.

When the device no longer needs memory, it calls: dma\_free\_coherent(); The CMA framework unmaps and releases memory back to the pool.

The core APIs in CMA are:

void \*dma\_alloc\_coherent(struct device \*dev, size\_t size, dma\_addr\_t \*dma\_handle, gfp\_t gfp);

void dma\_free\_coherent(struct device \*dev, size\_t size, void \*vaddr, dma\_addr\_t dma\_handle);

Internally, dma\_alloc\_coherent() → cma\_alloc() → alloc\_contig\_range() → low-level page allocator. If it fails, alloc\_contig\_range() invokes the page migration mechanism.

Pros of CMA

* Guarantees large contiguous memory allocations.
* Efficient use of memory by migrating movable pages.
* Allows dynamic resizing of CMA memory if unused.

Cons of CMA

* Cannot guarantee 100% success if memory is heavily fragmented.
* Reserved memory is wasted if the device never uses it.
* Page migration can be expensive if memory is highly fragmented.

DMA-BUF is a Linux kernel framework that enables buffer sharing between different devices without data copies. It works by exporting a shared memory buffer as a file descriptor (fd). Any device can import the fd and directly access the same buffer.

Without DMA-BUF, each device would need a separate memory copy. With DMA-BUF, multiple devices can share the same physical memory. Common use cases:

* + Camera → Display → Video Encoder sharing same buffer.
  + GPU → Display sharing same frame buffer.

DMA-BUF Flow in Kernel

* + Exporter Device (like Camera): Allocates memory using dma\_alloc\_coherent(). Exports a buffer as FD using: dma\_buf\_export();
  + Importer Device (like Display): Receives the file descriptor. Maps the buffer to device using: dma\_buf\_map\_attachment();
  + Both devices now directly access the same physical memory without copying.

dma\_buf\_unmap\_attachment() → Unmap the buffer.

Advantages of DMA-BUF

* Zero copy memory sharing.
* Efficient for high-performance multimedia.
* Common buffer management across devices.

|  |  |  |
| --- | --- | --- |
| Step | Camera Driver (Producer) | Display Driver (Consumer) |
| Step 1 | Export buffer using DMA-BUF | Import buffer using fd |
| Step 2 | Capture data in buffer | Render data from buffer |
| Step 3 | Free buffer after use | Unmap buffer after use |

|  |  |
| --- | --- |
| API | Description |
| cma\_alloc() | Allocates contiguous memory from CMA. |
| cma\_release() | Releases CMA memory. |
| alloc\_contig\_range() | Allocates a contiguous physical range. |
| dma\_alloc\_coherent() | Allocates DMA-coherent memory (uses CMA internally). |

In device tree, you can define reserved memory like:

reserved-memory {

cma\_region: cma@0 {

compatible = "shared-dma-pool";

reg = <0x0 0x80000000 0x0 0x20000000>;

reusable;

};

};

Use /proc/meminfo or /sys/kernel/debug/cma to monitor CMA usage:

cat /proc/meminfo | grep Cma

|  |  |
| --- | --- |
| Aspect | CMA Behavior |
| Purpose | Provides large physically contiguous memory to devices (like GPUs, cameras, etc.) |
| Allocation | Uses cma\_alloc() or dma\_alloc\_coherent() |
| Memory Management | Uses page migration, buddy allocator, and movable zones |
| Failure Case | Allocation fails if unmovable pages block the region |
| Configuration | Via kernel cmdline or device tree |

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | Write Data to Memory? | Mark Cache as Invalid? | Use Case |
| Flush (Clean) | ✅ Yes (if modified) | ❌ No | Before DMA read, process switch, ensuring memory has latest data |
| Invalidate | ❌ No | ✅ Yes | After DMA write, prevent stale cache data |
| Flush + Invalidate | ✅ Yes (if modified) | ✅ Yes | Before hardware handover, ensuring memory consistency |

Interview Pro Tip

* Always flush before invalidation if your data is modified and needs to be shared with another device (like DMA). Think case where I was copying chunks of IHM buffers from host to device in LBX and at start of each transfer, I had to flush it to memory so that after proper verification of image headers, it gets stored in proper memory correctly otherwise subsequent buffer download cache invalidation might corrupt validated previous buffer content in cache. Later, in when another image buffer gets copied from host using PCIe using DMA, then cache has to be invalidated so that memory content copied at RAM is visible correctly to CPU by cache.
* Invalidation without a flush may cause data loss if cache lines were dirty (modified but not written back).
* ARM architectures often combine flush + invalidate to ensure cache coherence.

|  |  |  |
| --- | --- | --- |
| Feature | Prefetchable BAR | Non-Prefetchable BAR |
| Caching | Allowed | Not Allowed |
| Speculative Reads | Allowed | Not Allowed |
| Usage | GPU RAM, SSD memory | Control registers |
| Mapping in Linux | ioremap\_wc() | ioremap() |

|  |  |
| --- | --- |
| Concept | Details |
| PCIe Cache Coherency Issue | CPU cache and PCIe DMA can have stale data. |
| Coherent DMA | No need for manual cache maintenance. |
| Non-Coherent DMA | Requires explicit cache flushing and invalidation. |
| Linux DMA API | dma\_alloc\_coherent(), dma\_sync\_single\_for\_cpu(), etc. |
| IOMMU | Helps maintain coherency on supported platforms. |

|  |  |
| --- | --- |
| Scenario | Use |
| Mapping a PCIe BAR | pci\_iomap() |
| Mapping a generic MMIO register | ioremap() |
| Mapping firmware regions (e.g., ACPI tables) | ioremap() |
| Mapping a GPU framebuffer | ioremap\_wc() |

|  |  |  |
| --- | --- | --- |
| Direction | Purpose | Example |
| Outbound (OB) | CPU to PCIe | Mapping CPU memory to PCIe device |
| Inbound (IB) | PCIe to CPU | Mapping PCIe device memory to CPU |

|  |  |  |
| --- | --- | --- |
| Feature | BAR Match Mode | Address Match Mode |
| Use Case | Inbound transactions (PCIe device to CPU) | Both Inbound & Outbound transactions |
| Matching Criteria | Matches a PCIe BAR number | Matches a specific address range |
| Best for | Memory-mapped BAR regions | Direct memory-to-memory translations |
| Example | Mapping BAR0 of a PCIe NIC to system RAM | Mapping CPU address 0x80000000 to PCIe device 0x50000000 |

|  |  |  |  |
| --- | --- | --- | --- |
| Stage | Component | Runs At | ACPI Dependency |
| BL1 | BootROM | EL3 (Secure) | No ACPI, only loads BL2 |
| BL2 | TFA (Loader) | EL3 (Secure) | No ACPI, loads BL31 |
| BL31 | TFA Runtime | EL3 (Secure) | Implements PSCI (used by ACPI for power management) |
| BL32 (Optional) | Secure OS (OP-TEE) | EL1 (Secure) | No direct ACPI, used for security services |
| BL33 | UEFI (Tianocore/EDK2) | EL2/EL1 | Passes ACPI tables (or DTB) to Linux |
| Linux Kernel | OS | EL1/EL2 | Uses ACPI tables for power management, PCIe, and hardware config |