**Contiguous Memory Allocator (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. Typical examples include:
  + Framebuffers for display controllers.
  + Camera buffers for video capture.
  + GPU buffers for texture processing.
* **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. The reservation size is defined through kernel parameters like:

cma=256M@0x88000000

Without CMA, the kernel would struggle to allocate large physically contiguous memory after the system has been running for some time.

**CMA Memory Allocation Flow**

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.

**2. ION (Android Memory Manager)**

**ION** is a memory management framework developed by **Android** to efficiently share memory between devices like GPU, display controllers, and video encoders. ION builds on top of **DMA-BUF** but provides an Android-specific interface for buffer management.

Suppose the camera captures a frame. The display controller needs the same buffer for rendering, and the video encoder needs it for compression. Without ION, multiple memory copies would be required, resulting in high latency and memory consumption. With ION, the same memory buffer can be shared between devices without copies.

**ION Heaps** are different memory pools used by ION to allocate memory. The key heaps are:

* + **System Heap:** Allocates normal memory.
  + **CMA Heap:** Uses CMA for contiguous memory.
  + **Carveout Heap:** Uses pre-reserved memory (like CMA region).
  + **Secure Heap:** Allocates memory for DRM.

**ION Buffer Allocation Flow**

User requests a buffer through /dev/ion: ion\_alloc\_fd(fd, size, flags);

ION searches for the appropriate heap. -> ION allocates memory using CMA (for large buffers) or normal pages. -> The buffer is represented as a **file descriptor (fd)**. -> This **fd** can be passed to other devices using DMA-BUF.

**ION Buffer Sharing Flow**

Camera captures a frame and requests memory. -> ION allocates memory from the CMA heap. -> The buffer FD is passed to the display driver. -> The display driver uses the same physical memory without copying.

**Pros of ION**

* Zero-copy buffer sharing between devices.
* Supports multiple heaps for different memory use cases.
* Prevents redundant memory copies.

**Cons of ION**

* Android-specific, not upstream in Linux kernel.
* Complicated heap management.

**3. DMA-BUF (Direct Memory Access Buffer)**

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.

**Senior Engineer Interview Questions**

**CMA Interview Questions**

1. **Can you dynamically increase CMA region size at runtime?**
2. **What is the role of alloc\_contig\_range() in CMA?**
3. **Why is contiguous memory important for GPUs or cameras?**

**ION Interview Questions**

1. **What are different ION heaps, and how do they differ?**
2. **How does ION use DMA-BUF under the hood?**
3. **Why is ION not upstreamed in the mainline Linux kernel?**

**DMA-BUF Interview Questions**

1. **Explain how DMA-BUF allows zero-copy buffer sharing between devices.**
2. **What is the role of dma\_buf\_get() and dma\_buf\_attach()?**
3. **How do you prevent buffer leaks in DMA-BUF?**
4. **Why is DMA-BUF critical for high-performance multimedia systems?**
5. **Explain a real-world scenario where DMA-BUF improves performance.**

example of how a device driver can allocate memory using **CMA**:

**my\_cma\_driver.c**

#include <linux/module.h>

#include <linux/platform\_device.h>

#include <linux/dma-mapping.h>

#define CMA\_SIZE 1024 \* 1024 \* 10 // 10 MB

static struct device \*my\_device;

static void \*cma\_mem;

static dma\_addr\_t cma\_handle;

static int my\_probe(struct platform\_device \*pdev)

{

my\_device = &pdev->dev;

// Allocate 10MB of contiguous memory

cma\_mem = dma\_alloc\_coherent(my\_device, CMA\_SIZE, &cma\_handle, GFP\_KERNEL);

if (!cma\_mem) {

dev\_err(my\_device, "CMA allocation failed\n");

return -ENOMEM;

}

dev\_info(my\_device, "CMA memory allocated: %pa\n", &cma\_handle);

return 0;

}

static int my\_remove(struct platform\_device \*pdev)

{

// Free the CMA memory

dma\_free\_coherent(my\_device, CMA\_SIZE, cma\_mem, cma\_handle);

dev\_info(my\_device, "CMA memory freed\n");

return 0;

}

static struct platform\_driver my\_driver = {

.probe = my\_probe,

.remove = my\_remove,

.driver = {

.name = "my\_cma\_driver",

},

};

module\_platform\_driver(my\_driver);

MODULE\_LICENSE("GPL");

**Explanation:**

* dma\_alloc\_coherent() allocates **contiguous memory** from the CMA region.
* dma\_free\_coherent() releases the memory.
* The physical address (cma\_handle) can be used by the hardware for DMA.

**B. Verify CMA Region in Kernel**

1. Boot the kernel.
2. Check available CMA regions:

cat /proc/meminfo | grep Cma

1. Expected output:

CmaTotal: 262144 kB

CmaFree: 258048 kB

1. You can also see the physical address assigned using dmesg:

dmesg | grep CMA

**C. Custom CMA Region via Device Tree**

If you want to use a specific CMA region via Device Tree:

reserved-memory {

cma\_region: cma@0x88000000 {

compatible = "shared-dma-pool";

reg = <0x88000000 0x1000000>;

reusable;

label = "cma\_region";

};

};

**2. ION (Android Memory Manager) Example**

**A. User-Space Application to Allocate ION Buffer**

This example uses ION to allocate memory in user space.

**my\_app.c (User-Space)**

c

CopyEdit

#include <fcntl.h>

#include <sys/ioctl.h>

#include <linux/ion.h>

#include <unistd.h>

#define ION\_HEAP\_SYSTEM 25

#define BUFFER\_SIZE 1024 \* 1024

int main()

{

int ion\_fd = open("/dev/ion", O\_RDWR);

struct ion\_allocation\_data alloc\_data;

int dma\_fd;

// Request ION buffer

alloc\_data.len = BUFFER\_SIZE;

alloc\_data.heap\_id\_mask = (1 << ION\_HEAP\_SYSTEM);

alloc\_data.flags = 0;

ioctl(ion\_fd, ION\_IOC\_ALLOC, &alloc\_data);

// Share buffer using fd

dma\_fd = alloc\_data.fd;

printf("Allocated DMA buffer fd: %d\n", dma\_fd);

close(dma\_fd);

close(ion\_fd);

return 0;

}

**Explanation:**

* Opens /dev/ion to request memory.
* Uses ION\_IOC\_ALLOC ioctl to allocate memory from the system heap.
* Returns a **file descriptor (fd)** that can be shared with other devices.

**B. Device Driver Consuming ION Buffer**

If you want a device driver to consume the buffer:

**my\_ion\_driver.c**

#include <linux/dma-buf.h>

int import\_ion\_buffer(int dma\_fd)

{

struct dma\_buf \*dmabuf;

// Import the file descriptor as a DMA buffer

dmabuf = dma\_buf\_get(dma\_fd);

if (IS\_ERR(dmabuf))

return PTR\_ERR(dmabuf);

// Map the buffer into kernel space

struct dma\_buf\_attachment \*attach;

attach = dma\_buf\_attach(dmabuf, my\_device);

// Use the buffer...

dma\_buf\_detach(dmabuf, attach);

dma\_buf\_put(dmabuf);

return 0;

}

**Explanation:**

* Uses dma\_buf\_get() to get a reference to the buffer.
* Maps the buffer into kernel space for device usage.
* Detaches and releases the buffer after use.

**3. DMA-BUF (Direct Memory Access Buffer) Example**

**A. Exporting DMA-BUF from a Driver (Producer)**

Suppose a camera driver captures a frame and shares it:

**my\_camera\_driver.c**

#include <linux/dma-buf.h>

static struct dma\_buf \*my\_dmabuf;

static int camera\_capture\_frame(void)

{

struct dma\_buf\_export\_info exp\_info = {

.size = 1024 \* 1024,

.flags = O\_RDWR,

.priv = NULL,

};

// Export DMA-BUF

my\_dmabuf = dma\_buf\_export(&exp\_info);

if (IS\_ERR(my\_dmabuf))

return PTR\_ERR(my\_dmabuf);

// Return the fd to user space

return dma\_buf\_fd(my\_dmabuf, O\_CLOEXEC);

}

**B. Importing DMA-BUF in Another Driver (Consumer)**

The display driver imports and displays the buffer:

**my\_display\_driver.c**

#include <linux/dma-buf.h>

static int display\_render\_frame(int dma\_fd)

{

struct dma\_buf \*dmabuf;

struct dma\_buf\_attachment \*attach;

struct sg\_table \*sgt;

// Get the buffer from fd

dmabuf = dma\_buf\_get(dma\_fd);

attach = dma\_buf\_attach(dmabuf, my\_device);

sgt = dma\_buf\_map\_attachment(attach, DMA\_BIDIRECTIONAL);

// Use the buffer in hardware...

dma\_buf\_unmap\_attachment(attach, sgt, DMA\_BIDIRECTIONAL);

dma\_buf\_detach(dmabuf, attach);

dma\_buf\_put(dmabuf);

return 0;

}

**✅ Summary of Interaction**

| **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 |