ChatGPT

Memory Mapping and DMA in Linux Device Drivers (Linux 6.x) – Beginner to Advanced Tutorial

Introduction

In this tutorial, we'll walk through **Chapter 15: "Memory Mapping and DMA"** from *Linux Device Drivers (LDD3)* – updating concepts to Linux 6.x and expanding explanations for both beginners and advanced readers. We'll cover how Linux manages memory for drivers, how to map device or kernel memory into user space, how to access user memory from the kernel, and how to perform Direct Memory Access (DMA) for high-performance I/O. By the end, you should understand:

- The Linux memory model relevant to drivers (logical vs. virtual vs. physical addresses, DMA addresses, high vs. low memory).
- Key kernel structures like `mm_struct` (process memory descriptor) and `vm_area_struct` (memory region descriptor), and how the `mmap()` system call is implemented for device drivers.
- Two approaches to driver memory mapping: using `remap_pfn_range()` for contiguous physical memory vs. providing a page **fault handler** for dynamic or noncontiguous mappings, and when to use each.
- How to map user-space memory into the kernel (pinning pages with `get_user_pages()` or newer `pin_user_pages()` APIs) for direct I/O, including marking pages dirty (`SetPageDirty`) and releasing them (`put_page`, formerly `page_cache_release`) [1] [2].
- The principles of **Direct I/O** (bypassing kernel buffering) and **Asynchronous I/O** (AIO) as they apply to driver design – including Linux's AIO interfaces and modern alternatives.
- DMA basics and the Linux DMA API: allocating DMA-coherent memory (`dma_alloc_coherent`), creating streaming DMA mappings (`dma_map_single`, `dma_map_page`, etc.), cache coherency considerations (coherent vs. streaming DMA), scatter/gather DMA with scatterlists, DMA pools for small buffers, 64-bit addressing (PCI DAC), and legacy ISA DMA with the 8237 DMA controller.

Along the way we include **step-by-step diagrams** and code snippets to illustrate critical flows like address translation, memory mapping, and DMA synchronization. Beginner-friendly sections explain the "what" and "why" in simple terms, while advanced commentary dives into kernel internals and edge cases (marked as **Advanced**). In the final section, we present a hands-on project for Raspberry Pi (64-bit) – implementing a character driver that uses DMA-safe memory and `mmap` to share a buffer with userspace, simulating a simple DMA data transfer.

Let's get started with the fundamental concepts of Linux memory management as they relate to writing device drivers.

Linux Memory Model and Address Types (Driver Perspective)

Linux, like all modern OSes, uses **virtual memory** – a layer of address translation between what programs see and physical RAM addresses. As a driver developer, you must be mindful of the different address forms and memory regions Linux uses [3] [4]. Here are the key address types and terms:

- **User Virtual Addresses:** The addresses used by user-space programs. Each process has its own virtual address space (e.g. 0x00000000 to 0xbfffffff on 32-bit with a 3G/1G split, or a much larger range on 64-bit). These addresses are translated by the CPU's Memory Management Unit (MMU) via page tables into physical addresses [5]. A user address is valid only in the context of its process. For example, pointer `0x7ffdf000` in one process refers to different physical memory than the same address in another process.

- **Physical Addresses:** Actual addresses on the memory bus seen by RAM chips. Physical memory is divided into fixed-size **pages** (commonly 4KB). The MMU and page tables map virtual pages to physical frames [6] [7]. Drivers rarely use raw physical addresses directly, because the kernel operates in virtual address space. However, physical addresses matter for hardware DMA and memory-mapped I/O. (Physical addresses are typically 32-bit on older systems, 64-bit on modern systems, though a 32-bit CPU with Physical Address Extension can use more physical memory).

- **Bus Addresses:** Addresses used by peripheral buses (PCI, ISA, etc.) to access memory [8]. Often the bus address is the same as the physical address on simple architectures, but that's not guaranteed. Some systems have an IOMMU (I/O Memory Management Unit) or offsets that make the bus view of memory different. For example, a PCI device might use bus address 0xF0000000 which the IOMMU maps to physical 0x3F0000000. **Devices perform DMA using bus addresses**, not CPU virtual addresses [9] [10]. The Linux DMA API abstracts these differences, providing functions to obtain a DMA (bus) address for a given buffer.

- **Kernel Logical Addresses:** These are the normal addresses the kernel uses in its own address space (for direct-mapped RAM). On most architectures, the kernel "logical" address space is a **linear mapping of a certain portion of physical memory** (usually the lower physical addresses) [11]. For instance, on a 32-bit x86 with 1GB kernel space, physical addresses 0x00000000 through 0x3FFFFFFF might be mapped straight to virtual addresses 0xC0000000 through 0xFFFFFFFF (offset by a constant). This region is often called **"low memory"** – memory that the kernel can address directly with a pointer [12] [13]. Memory allocated by `kmalloc()` comes from this direct-mapped region and can be accessed via ordinary pointers (type `void *` or `unsigned long`). Kernel logical addresses are convenient but exist only for the "low" physical memory range (the exact cutoff depends on architecture and kernel configuration).

- **Kernel Virtual Addresses:** Kernel virtual addresses encompass all addresses the kernel can use, including logical addresses and those from high-memory mappings. Some kernel memory (like memory allocated by `vmalloc` or ioremap) does **not have a direct physical mapping** – instead, it's set up via page tables on demand. These are still kernel-addressable (you can use them as pointers in kernel code), but they are **not linear mappings of physical memory** [14]. For example, `vmalloc()` might give a contiguous range of kernel virtual addresses that actually maps to noncontiguous physical pages behind the scenes. Similarly, `kmap()` (described below) creates

temporary kernel virtual mappings for “high memory” pages. All kernel logical addresses are also kernel virtual addresses, but not vice versa [15].

High Memory vs. Low Memory (and kmap)

On 32-bit systems, the kernel cannot keep the entire physical RAM directly mapped because of address space limits. The portion it can directly map is called **low memory**, and any physical memory above that limit is **high memory** [13]. High memory pages have no kernel logical address – meaning the kernel can’t directly dereference them without setting up a mapping. Drivers must be aware of this when dealing with memory that might reside in high memory (e.g., user pages on a system with lots of RAM, or buffers allocated without GFP_HIGHUSER moves).

To access a high-memory page, the kernel provides `void *kmap(struct page *page)` which **maps a high page into the kernel’s address space (temporarily)** and returns a pointer [16]. You must call `kunmap(page)` when done. On low-memory pages, `kmap` may simply return the normal logical address (as low pages are always mapped) [17]. High memory isn’t an issue on most 64-bit systems – they have a vast virtual address space that can map all physical memory, so the concept of “highmem” is mostly relevant to older 32-bit machines. (In fact, on Raspberry Pi 4 64-bit, all 8GB RAM can be mapped by the kernel directly, so `kmap` is essentially a no-op or not needed.) We mention it because drivers in the kernel still contain conditional code for highmem, and an awareness helps in understanding older APIs and bounce buffering (discussed later).

Advanced: High memory pages cannot be used for DMA with devices that lack an IOMMU or 64-bit addressing, because the device cannot address those high physical addresses [18]. The kernel may need to **bounce** such pages through a low-memory buffer for DMA (explained in DMA section). Also, certain legacy interfaces (like some old drivers or `/dev/kmem`) expect pages to be marked “reserved” if they are permanently removed from normal paging – historically, drivers would call `SetPageReserved` on pages they are managing manually (such as pages from `alloc_pages` that they map to user space) [19] [2]. Modern kernels have mostly repurposed the “reserved” flag (and many usages are gone), so drivers now rely on proper VM flags (like `VM_IO` for memory-mapped I/O regions) to prevent the kernel from swapping or messing with those pages.

Page Structures and Memory Zones

For every physical page frame in RAM, Linux maintains a `struct page` descriptor (typically in a global array called `mem_map`). This struct stores metadata: flags (locked, dirty, reserved, etc.), reference count, mapping info, and so on. Drivers often encounter `struct page` when dealing with higher-level memory management (for example, `get_user_pages()` returns an array of `struct page*`). Some useful utilities: `pfn_to_page(pfn)` gives the page struct for a physical frame number [20], `virt_to_page(kaddr)` gives the page for a low-memory kernel address, and `page_to_virt(page)` (or `page_address(page)`) returns the kernel logical address if the page is in low memory [21]. If the page is high memory, `page_address` may be NULL (meaning you must kmap it first).

Linux divides physical memory into zones for allocation: e.g. `ZONE_DMA`, `ZONE_DMA32`, `ZONE_NORMAL`, `ZONE_HIGHMEM` (the last one only on 32-bit with highmem). For example, `ZONE_DMA` might represent memory < 16MB suitable for ISA DMA, and `ZONE_DMA32` is the first 4GB for devices limited to 32-bit DMA.

Typically, normal allocations come from `ZONE_NORMAL` (all directly mapped RAM on 64-bit). If a driver needs memory in a specific zone, it can use GFP flags (e.g. `GFP_DMA`) to restrict allocation to that zone [22]. This is mostly relevant for legacy devices (ISA or certain bus mastering limitations); more on this in the DMA section.

Summary (Beginners): Think of the system as having multiple "views" of memory: user programs have their own private virtual address view, the kernel has its own (with a part directly mapping physical memory), and devices have yet another view (bus addresses). Translating between these views is a common task in driver programming – e.g., translating a user pointer to a kernel address or ensuring a kernel buffer's physical address is provided to a device for DMA. Always be clear which address type you're dealing with in a driver API. If unsure, check documentation – many bugs come from using the wrong address (for instance, giving a device a CPU virtual address instead of a bus address will not work [23] [24]).

Advanced: The kernel does not use distinct C types for the various address spaces, so it's up to the programmer to keep track [25]. Some common conversions: `__pa()` to get the physical address of a kernel logical address (valid only for low memory) [26], and `__va()` to get the kernel logical address from a low physical address [26]. However, in driver code you'll more often use safe APIs (like the DMA API or vmalloc/vmap functions) rather than these low-level macros, to handle tricky cases like high memory or IOMMUs portably.

Finally, let's introduce two key structures in Linux memory management:

- `struct mm_struct`**:** this is the memory descriptor for a process (also called an "address space" or "memory mm"). It contains the list of VMAs, pointers to page tables, and other info about the process's memory layout. Each user process has one `mm_struct` (shared by threads). Kernel threads have no `mm_struct` (they use kernel address space only). Drivers typically don't manipulate `mm_struct` directly, but when implementing `mmap`, the kernel will pass you a `struct vm_area_struct` which has a pointer to the `vm_mm` (so you know which process is mapping your device) [27] [28].

- `struct vm_area_struct` **(VMA):** this struct represents a continuous region of virtual memory in a process, with uniform properties (permissions, whether it's backed by a file or device, etc.). The kernel maintains a sorted list (and tree) of a process's VMAs (see `/proc/<pid>/maps` for a human-readable dump). Each VMA has fields like `vm_start` (start address), `vm_end` (end address), `vm_page_prot` (page protection flags), `vm_flags` (attributes like VM_IO, VM_DONTCOPY, etc.), `vm_file` (file pointer if this VMA maps a file/device), `vm_pgoff` (offset into the file or device region), and `vm_ops` (callbacks for special handling) [29] [30]. When a user calls `mmap()`, the kernel creates a new VMA (or repurposes an existing one) to cover the requested range and calls the file's `mmap` method (driver hook) to setup the mapping.

Understanding VMAs is crucial for driver memory mapping: the driver's `mmap` function is handed a `struct vm_area_struct *vma` and should **associate the VMA with the physical or virtual addresses of the device or memory** it wants to map. We'll see how to do that next.

Implementing mmap in a Driver (Mapping Kernel/Device Memory to User Space)

The `mmap()` system call allows user-space to map a portion of a file or device into its address space, so it can access it like memory. For device drivers, implementing `mmap` is optional but powerful – it enables **zero-copy** data transfers and direct user access to device memory, which can significantly improve performance for some hardware. Not all drivers need `mmap`, but consider it if your device handles large buffers or memory regions (e.g., framebuffers, camera buffers, PCI device memory, etc.).

In the driver, `mmap` is a file operation (in your `struct file_operations`). The prototype is:

```
int mydriver_mmap(struct file *filp, struct vm_area_struct *vma);
```

When a user process calls `mmap(fd, length, prot, flags, offset)`, the VFS will route it to your driver's `mydriver_mmap`, passing in the VMA representing the user's requested memory region. Your job is to tie this VMA to the physical or kernel memory that should back it. There are two common techniques to implement device memory mapping:

Approach 1: Direct Remapping with `remap_pfn_range` (Contiguous Physical Memory)

If the memory to map is a **contiguous range of physical addresses** (such as device memory or a continuous chunk of system RAM you allocated), you can use `remap_pfn_range()` to map it directly into user space. This function essentially tells the kernel, "for this VMA, use these physical page frames." It avoids the normal page-by-page faulting mechanism by setting up all the PTEs (page table entries) in one go.

Usage: `int remap_pfn_range(struct vm_area_struct *vma, unsigned long user_addr, unsigned long pfn, unsigned long size, pgprot_t prot);`

- `user_addr` is usually `vma->vm_start` (the beginning of the VMA in user space).
- `pfn` is the starting **physical frame number** of the memory you want to map (i.e. physical_address >> PAGE_SHIFT).
- `size` is the length of the area (page-aligned).
- `prot` is the memory protection for the mapping (typically you use `vma->vm_page_prot`, possibly adjusted for cache attributes).

For example, say your device has memory at physical address 0xFE000000 of length 0x1000 bytes that you want user to access. In `mydriver_mmap`, you might do:

```
vma->vm_page_prot = pgprot_noncached(vma->vm_page_prot);  // if device memory,
map uncached
unsigned long pfn = 0xFE000000 >> PAGE_SHIFT;
if (remap_pfn_range(vma, vma->vm_start, pfn, 0x1000, vma->vm_page_prot)) {
    return -EAGAIN; // mapping failed
```

```
}
return 0;
```

This maps the device's memory into the process. After this call, the process's addresses `[vma->vm_start, vma->vm_end)` correspond to physical addresses starting at 0xFE000000. The user can simply dereference the memory or use `memcpy`, etc., and it will read/write directly to the device's memory. No read() or write() system calls needed in that path.

A similar scenario is if you have a buffer from `kmalloc` or `get_free_pages` that you want to share. **However, caution:** normal kernel allocations are in *kernel virtual* memory, not necessarily contiguous physical memory if larger than a page. If you need to map a larger physically-contiguous buffer, you might have to ensure you allocated it with `alloc_pages` of an adequate order or used CMA (Contiguous Memory Allocator) or dma_alloc_coherent (which often gives contiguous memory) – more on allocation in DMA section. If the memory is not physically contiguous, you can't directly use one `remap_pfn_range` over the whole range (you'd need to map page by page or use the fault method described next). For small buffers that fit in one page, it's fine.

Reserved vs Non-reserved Pages: In older documentation, you'll read that only "reserved" pages can be remapped to userspace safely [31]. "Reserved" in this context meant pages not managed by the normal VM (e.g., device memory or boot memory). The concern was that the VM shouldn't swap them or free them unexpectedly. In modern Linux, the behavior is controlled by flags on the VMA (`VM_IO`, `VM_PFNMAP`) which you should set for such mappings. The kernel will then treat those pages as I/O memory. When using `remap_pfn_range`, the kernel automatically sets `VM_IO` and `VM_PFNMAP` in the VMA if it's not a RAM file mapping. If you're mapping normal RAM pages that the kernel allocated, you should also set `VM_DONTEXPAND` (to prevent mremap from extending it) and consider `VM_DONTDUMP` (to exclude from core dumps). Example:

```
vma->vm_flags |= VM_IO | VM_DONTEXPAND | VM_DONTDUMP;
```

Setting the proper flags communicates the special nature of this mapping to the kernel (for instance, `VM_IO` regions are not included in process memory core dumps and are exempt from random fodder like KSM).

Cache coherency: If you are mapping device memory or memory that isn't cached, you likely want to use `pgprot_noncached()` on the VMA's page prot [32], as shown above. This macro adjusts the page attributes to uncached (or writes combined, etc., depending on your needs). This prevents CPU caching on that memory range, which is usually necessary for memory-mapped device registers or DMA buffers to avoid stale data issues. On some architectures (x86), device memory is strongly ordered and not cached by default when using ioremap; but on others (ARM), you must explicitly request an uncached mapping. Always consider if the region needs to be cached or not – typically, framebuffers might use write-combining, control registers use uncached, etc.

Error handling: If `remap_pfn_range` returns nonzero, the mapping failed (likely out of resources). You should then return an `-EAGAIN` or `-EFAULT` from your `mmap` function. If it returns 0 (success), you

return 0. After success, any access by the user will directly map to the pages you specified; no page faults for those pages will come to your driver (they are pre-populated in the process page table).

This approach is straightforward for simple contiguous memory. Many drivers use it to map device memory (e.g., PCI BARs) to user space. For instance, `/dev/mem` (the driver that allows raw physical memory access) uses `remap_pfn_range` internally to map arbitrary physical pages to user space [33].

One limitation: **the memory must not be already in use by the kernel or have "regular" page structs that the kernel might swap**. If you map normal pageable RAM this way without telling the kernel, it could lead to inconsistencies. That's why either the pages should be allocated specifically for this purpose and marked accordingly or you should use the second approach (fault-based mapping) which integrates with VM paging.

Approach 2: On-Demand Paging with vm_ops->fault (Noncontiguous or Managed Memory)

If the memory you want to expose is not a nice contiguous physical chunk – for example, an array of pages scattered in memory, or you only want to allocate backing pages on demand – you should implement a **page fault handler** for the VMA instead of mapping everything at once. In older kernels, drivers would provide a `nopage()` callback on `vm_ops` for this; in Linux 6.x, the `nopage` callback is long deprecated in favor of the `fault` callback [34].

Using a fault handler means you don't populate PTEs in `mmap` directly. Instead, you set up `vma->vm_ops` to point to a `struct vm_operations_struct` with a `.fault` handler. Then return 0 from `mydriver_mmap` (so the VMA is created). When the process actually tries to access a page in that VMA, the CPU triggers a page fault (because initially no physical page is attached). The kernel will invoke your `.fault` handler, which is responsible for *finding or allocating the physical page for that address* and installing it into the process's page table.

This method is more complex but offers flexibility: you can decide on a per-page basis what to map. For instance, if a user mmap's a 10 MB region but only accesses a few KB, you might allocate and map pages lazily for those KB (demand allocation). Or the pages might come and go (for example, if implementing an ephemeral buffer or an interface to high memory where you swap pages in).

How to implement: In `mydriver_mmap`, do something like:

```
vma->vm_ops = &mydriver_vm_ops;
vma->vm_flags |= VM_IO | VM_DONTEXPAND | VM_DONTDUMP;
return 0;
```

And have a global `struct vm_operations_struct mydriver_vm_ops = {`
`    .fault = mydriver_vma_fault,`
`    .open = mydriver_vma_open,    // optional`
`    .close = mydriver_vma_close   // optional`
`};`

The `open` and `close` can be used to track how many processes have the mapping, etc., but they are optional.

Your `mydriver_vma_fault` will be called with `struct vm_fault *vmf` (which contains `vmf->address` – the faulting user address, and `vmf->vma`, etc.). The job is to **find the corresponding physical page** for that address and map it. The simplest way in modern kernels is to use `vm_insert_page` or `vm_insert_pfn`. For example:

```
static vm_fault_t mydriver_vma_fault(struct vm_fault *vmf) {
    struct vm_area_struct *vma = vmf->vma;
    unsigned long address = vmf->address;  // Faulting virtual address in user
space
    unsigned long offset = address - vma->vm_start + ((uint64_t)vma->vm_pgoff
<< PAGE_SHIFT);
    // offset is how far into our device/file this address is.
    struct page *page = my_lookup_or_alloc_page(offset);
    if (!page)
        return VM_FAULT_OOM;
    // got the page (with refcount incremented if newly allocated)
    int err = vm_insert_page(vma, address, page);
    if (err) {
        put_page(page); // release if not inserted
        return VM_FAULT_SIGBUS;
    }
    return VM_FAULT_NOPAGE;
}
```

Here, `my_lookup_or_alloc_page(offset)` is pseudo-code where you translate the offset to your backing storage. For example, if you have an array of pages already allocated, you index into it. Or if this is device memory at some physical base, you create a page struct from the physical frame (with `pfn_to_page`). **Important:** If you use `alloc_page` to create a new page, you should account for freeing it later (perhaps when the VMA is closed, or implement `.close` to clean up). If it's a persistent mapping tied to device lifetime, the pages might remain until device close.

After calling `vm_insert_page`, the VM takes over the page's ownership in that VMA and will handle the PTE installation. The function `vm_insert_page` will increase the page refcount and associate it with the VMA. If the page is already mapped (multiple faults on same page), it will return `-EBUSY` (which would be unusual here unless user is racing faults). Typically you return `VM_FAULT_NOPAGE` to indicate success (the naming is historical: VM_FAULT_NOPAGE means "no page was originally present, we filled it now, and no special action needed").

This approach can handle noncontiguous pages easily – each fault can map a completely unrelated page. It's also how you would map high memory pages or pages that should stay under kernel management. For instance, if you have data in page cache or some existing kernel pages, you can just insert them.

Example use case: Suppose you have a large buffer that was vmalloc'ed in kernel (which is non-physically-contiguous). You can't use `remap_pfn_range` for the whole range because vmalloc gives noncontiguous frames. Instead, you can store the list of pages from vmalloc (via `vmalloc_to_page` for each page in the range) and then in the fault handler, map the correct page for the offset. The LDD3 text suggests using the fault method for such cases [35]. This way, each vmalloc page is mapped into user on demand.

Managing the mapped pages: When the user process unmapps or exits, the kernel will automatically free the pages (decrement refcounts) if they were inserted via `vm_insert_page`. If you allocated them on the fly and want to be notified to potentially free them back to the system, you might use the `close` VMA op – it gets called when the VMA is destroyed. In `close`, you could free any remaining allocated pages (though be careful: if the process forked, multiple VMAs could point to the same pages; reference counts normally handle that). Another strategy is to allocate all needed pages at `mmap` time and free at driver release time – simpler but it defeats the demand paging benefit.

Comparison of remap vs fault method:
- *Remap upfront:* simpler to implement if you have contiguous memory or a predefined region. User gets immediate access without page faults for the range. But you must have the memory ready and you can't easily map noncontiguous sets (except by calling `remap_pfn_range` multiple times page by page, which is essentially doing what the fault handler would do, but all at once). Also, for very large mappings, prefaulting everything can consume memory unnecessarily if user won't use it all.
- *Fault on access:* more flexible and potentially memory-efficient. Can map pages on demand, and supports any collection of pages. But a bit more code complexity. Also, if hardware is providing data asynchronously, you might not want user to see a page until data is ready – a fault handler could block or wait until the page is filled, or you could prefault and then delay usage (design decision).

Advanced note: Modern kernels also have `vmf_insert_pfn()` and `vmf_insert_page()` which are variants to use inside fault handlers. For example, `vmf_insert_pfn(vma, address, pfn)` inserts a physical page frame without needing a `struct page` (useful for device memory where you might not have a struct page if it's IO memory). Use `vmf_insert_pfn_prot` if you need a specific pgprot. The return conventions differ slightly (they return VM_FAULT codes directly). It's essentially the newer way to do what vm_insert_page did but in the vm_fault context. You should check the latest API, but conceptually it's similar. The key is: **only call these insertion functions from within your fault handler**; do not call them from the initial `mmap` context (the VMA isn't fully setup for page insertion until fault time).

Example: Mapping Device Memory (PCI BAR) to User Space

To cement understanding, let's consider a concrete example. Suppose you have a PCI device with memory BAR at 0xd0000000 (physical address) of length 64KB, which is memory-mapped I/O for a frame buffer. You want to allow a user-space library to mmap this and draw directly. Here's how the driver might implement it:

- During probe, you ioremap the BAR: `dev->regs = ioremap(pci_resource_start(pdev, 0), 0x10000);` – this gives a kernel virtual address for the device registers or memory. (You might also enable the device and request regions, etc., omitted here.)

- In the `mmap` fops: You want to map that device memory to user. Because it's device memory (not regular RAM), we'll use `remap_pfn_range`. The physical start is 0xd0000000.

```
static int myfb_mmap(struct file *filp, struct vm_area_struct *vma) {
    unsigned long phys = 0xd0000000 + (vma->vm_pgoff << PAGE_SHIFT);
    unsigned long vsize = vma->vm_end - vma->vm_start;
    if (vsize > 0x10000)  // trying to map more than BAR
        return -EINVAL;
    // Set uncached or write-combining attribute for this VMA
    vma->vm_page_prot = pgprot_writecombine(vma->vm_page_prot);
    // Prevent core dumping and extension
    vma->vm_flags |= VM_IO | VM_DONTEXPAND | VM_DONTDUMP;
    if (remap_pfn_range(vma, vma->vm_start, phys >> PAGE_SHIFT, vsize, vma-
>vm_page_prot)) {
        return -EAGAIN;
    }
    return 0;
}
```

After this, user space can directly `mmap()` the device file and get access to the framebuffer memory. The use of `writecombine` allows faster sequential writes (important for framebuffers), while disallowing CPU read caching (so reads are somewhat uncached – which is fine if the CPU doesn't need to read from that memory often or if coherence isn't an issue).

- This example is relatively straightforward. Many drivers in the kernel do exactly this for memory BARs. Note that we used `vma->vm_pgoff` (the offset passed to mmap) to allow mapping offsets within the BAR. The user could pass an `offset` to mmap; the kernel sets `vma->vm_pgoff = offset >> PAGE_SHIFT`. We must account for that by adding it to the base physical address. In our check, we ensure the requested size doesn't exceed our resource.

Example: Mapping Driver-Allocated Buffer to User (Continuous DMA buffer)

Consider a driver that uses a DMA buffer for data transfer (perhaps a camera capturing into a ring buffer). The driver allocates, say, 8 pages of contiguous memory for the buffer (using `dma_alloc_coherent` or `alloc_pages` with `__GFP_COMP` to get 8 contiguous pages). This buffer has a kernel logical address `kbuf` and a physical address (for DMA) we can get via `virt_to_phys(kbuf)` (assuming contiguous). We want user to mmap this buffer to see incoming data in real-time.

We can implement `mmap` similar to above:

```
static int mydrv_mmap(struct file *filp, struct vm_area_struct *vma) {
    unsigned long size = vma->vm_end - vma->vm_start;
    if (size > BUFFER_SIZE)
        return -EINVAL;
    // Mark as shared DMA memory
    vma->vm_flags |= VM_IO | VM_DONTDUMP;
    // No caching if CPU-coherent needed; if dma_alloc_coherent used, memory is
already coherent or noncached.
```

```
    vma->vm_page_prot = pgprot_writecombine(vma->vm_page_prot);
    unsigned long phys_start = virt_to_phys(mydrv->kbuf);
    if (remap_pfn_range(vma, vma->vm_start, phys_start >> PAGE_SHIFT, size, vma-
>vm_page_prot))
        return -EAGAIN;
    return 0;
}
```

This will map the contiguous DMA buffer. Because we used `dma_alloc_coherent`, on many platforms that memory is marked as device coherent (noncached on non-coherent arch, or with special cache settings). Using `pgprot_writecombine` or `pgprot_noncached` matches how `dma_alloc_coherent` behaves (it might already be noncached, but an extra noncached setting won't hurt). Now user can treat the buffer as shared memory with the driver and possibly even perform some zero-copy operations. If the device DMA writes into this buffer, user sees the data change in memory directly.

Memory security: One must be careful – mapping kernel memory to user means user-space could potentially overwrite it (if permissions are PROT_WRITE) and thus affect kernel state. Only map buffers that are intended for user access and are **not holding confidential data or kernel structures**. For DMA buffers or device memory, this is fine as long as you trust the user or have appropriate permissions on the device node. Generally `/dev` devices rely on Unix file permission for security; once a privileged program mmap's, it could do bad things if the driver is not careful (like mapping the wrong memory). So always validate offsets and sizes to ensure you only map the intended physical memory.

TL;DR for beginners: To implement `mmap` in a driver, you either (a) directly map a physical region using `remap_pfn_range` (simple but requires contiguous memory or device memory), or (b) set up a page-fault callback to dynamically provide pages (more complex but flexible). Approach (a) is common for hardware memory; approach (b) is used for pseudo-devices or when managing normal RAM pages to user. Both require setting proper VMA flags and protections. Once done, user programs can use pointers to interact with hardware or buffers directly, without extra copy overhead.

Advanced considerations:
- If you map regular RAM pages (like those from `kmalloc` or page allocator) via remap, note that these pages are still "owned" by the kernel's allocator. Historically, drivers would call `SetPageReserved(page)` on each page to flag it out of the pager's reach [19]. In modern kernels, that's typically unnecessary or even removed – the VM_PFNMAP on the VMA is sufficient to prevent the VM from doing anything with them. But you must keep a reference to those pages (increment refcount) to ensure they don't get freed while user has them mapped. One way is to allocate them with `alloc_pages` and *not free them until the device closes* after unmapping. Another way is to pin them (`get_page` on each) and implement VMA close to put them. The fault-method using `vm_insert_page` automatically takes a ref, which is put on VMA destruction. With remap_pfn_range, the kernel doesn't automatically adjust refcount, so the driver must ensure the pages stay allocated. In practice, if you allocated the buffer and don't free it until user is done, it's fine.

    - The `access_ok` and `follow_pfn`: There was a function `follow_phys` in some contexts to handle pfn mapping in older kernels, but it's internal. Just be aware that mapping arbitrary physical addresses via `/dev/mem` or custom drivers can bypass certain kernel protections, so on secure

systems, CAP_SYS_RAWIO is needed (the driver .mmap can check credentials if necessary). Usually, device drivers assume the user has permission to open the device, and that suffices.

Now that we've covered mapping device memory to user space, let's look at the opposite: mapping user memory into the kernel for direct I/O.

Accessing User-Space Memory from Kernel (Direct I/O and `get_user_pages`)

Normally, when a user program issues a read or write, the driver uses `copy_to_user()` or `copy_from_user()` to transfer data between a kernel buffer and user buffer. This is **buffered I/O**: an intermediate copy. However, sometimes we want to avoid that copy for performance, or perhaps we want the device to DMA directly to the user's buffer. In such cases, the driver needs to **access or pin user-space memory** directly in kernel.

Use case: A user passes a pointer (and length) to the driver via an ioctl or write call, and the driver needs to fill that buffer with data from hardware. If the data is large, copying it twice (device->kernel, kernel->user) is inefficient. It would be nicer if the device could DMA straight into the user's pages. But the kernel can't just hand any user address to the device – it needs to resolve it to physical pages and lock them in memory (so they don't get swapped or moved while DMA is happening). That's what `get_user_pages()` **(GUP)** does: it takes an address range in a user process and pins the underlying physical pages, returning them for driver use [36].

`get_user_pages()` Basics

The classic function call (since Linux 2.6) is:

```
long get_user_pages(struct task_struct *tsk, struct mm_struct *mm,
                    unsigned long start, unsigned long nr_pages,
                    int write, int force, struct page **pages, struct
vm_area_struct **vmas);
```

This interface has evolved, and in 6.x there are wrappers like `get_user_pages_remote`, `get_user_pages_locked`, etc., and flags instead of boolean params. But conceptually: you provide the starting user address and number of pages, whether you want to write to these pages, and it will **pin** those pages in memory and put pointers to their `struct page` in the `pages` array. "Pin" here means the page's refcount is incremented to prevent it from being reclaimed or swapped. If the pages are not present, the function will page them in (and if `force` is 1, it will override RO protections to pin a read-only page for writing, used in O_DIRECT case with read-only file mapping). The function returns the number of pages pinned (or a negative error). After this, the driver can convert those struct page's to physical addresses (for DMA) or kmap them to access data, etc.

Important: After you're done, you must release the pages via `put_page()` (or historically `page_cache_release()`) for each page [1] [2]. Failing to do so leads to a memory leak (and the page remains "pinned", i.e., not movable or reclaimable, which is bad).

Also, if the driver writes to these pages (e.g., DMA from device fills them, or driver CPU writes data), you should mark them dirty if they were originally memory-mapped file pages. Setting `SetPageDirty(page)` ensures that if the page was from a file (like mmap of a file via O_DIRECT), the filesystem knows the page contents have changed and will eventually write back [1]. If you neglect this, the kernel might drop the page thinking it's clean (since it came from disk file and hasn't been marked dirty, it assumes disk copy is up-to-date), causing data loss. However, if pages are anonymous (heap memory) or already backed by swap, dirty marking is less crucial but still generally done for safety. Typically:

```
if (!PageReserved(page)) SetPageDirty(page);
```

The `PageReserved` check historically avoids marking special reserved pages (which have no backing store) dirty [37]. User pages are rarely reserved, so this is usually just always true. In modern code, you might omit this check; it doesn't hurt to dirty a normal anonymous page.

Let's demonstrate with a scenario:

Example: Directly DMA to User Buffer (O_DIRECT style) – Suppose our driver receives an ioctl with a user pointer to a buffer and a length, and the hardware will DMA data into that buffer. The sequence might be:

1. Use `get_user_pages` to pin the user pages containing that buffer. For example, if the buffer is 40KB and starts at a non-page-aligned address, GUP will pin the 10 pages or so that cover that range.

2. For each pinned page, get the DMA address by calling `dma_map_page` (streaming DMA mapping) with the page and offset within page [38]. Alternatively, some drivers might use `kmap` to map the page and then have the device PIO data into it, but DMA is more efficient.

3. Initiate the DMA transfer(s) to those addresses. Possibly use scatter-gather if device supports, since you have multiple pages.

4. Wait for DMA complete (interrupt).

5. Mark pages dirty if writing to them (since the device wrote new data into them) [1]. Also mark accessed if needed (not usually necessary).

6. Unmap the DMA addresses (`dma_unmap_page` for each) and release the pages (`put_page`). Only release after DMA complete, obviously.

7. The user can then see the data in its buffer.

If the device was reading from user memory (device DMA from user to device), you'd pin pages, `dma_map_page` with DMA_TO_DEVICE, perhaps flush/invalidate caches, then after DMA you might not need to mark dirty (since data was read, not written). But you'd still unpin pages after.

Example code snippet (simplified):

```
// Assume current->mm is the target mm and we are in process context.
unsigned long uaddr = arg.user_buf;
size_t len = arg.len;
int nr_pages = (len + (uaddr & ~PAGE_MASK) + PAGE_SIZE - 1) / PAGE_SIZE; //
buffer spanning pages
struct page **pages = kmalloc_array(nr_pages, sizeof(*pages), GFP_KERNEL);
int pinned = get_user_pages(uaddr, nr_pages, WRITE, 0, pages, NULL);
if (pinned < 0) {
    // handle error
}
// Now pages[0..pinned-1] are pinned.
for (i = 0; i < pinned; i++) {
    // DMA map each page (or use dma_map_sg if contiguous in user virt)
    dma_addr_t dma = dma_map_page(dev, pages[i], 0, PAGE_SIZE, DMA_FROM_DEVICE);
    if (dma_mapping_error(dev, dma)) { /* handle error, unmap previous, etc */ }
    dma_addrs[i] = dma;
}
// Now program device with dma_addrs list...
```

After DMA done (in IRQ handler or bottom half):

```
for (i = 0; i < pinned; i++) {
    dma_unmap_page(dev, dma_addrs[i], PAGE_SIZE, DMA_FROM_DEVICE);
    if (!PageReserved(pages[i]))
        SetPageDirty(pages[i]);
    put_page(pages[i]);
}
```

This pattern is essentially what the kernel's direct I/O (O_DIRECT) does under the hood for block devices: it gets user pages, maps for DMA, does I/O, dirty's and releases. If you're writing a char driver that wants to mimic that, this is the way.

Scatterlist note: Instead of mapping each page individually, you can populate a `struct scatterlist` array with each page and length, then call `dma_map_sg` once to map all at once (the DMA API might coalesce them if physically contiguous). That yields a list of DMA addresses and lengths the device can use. This is more efficient if the device supports scatter-gather DMA and you have many pages.

Accessing user memory from kernel without DMA: Even if you're not doing DMA, you might use `get_user_pages` to simply gain direct CPU access to user memory (e.g., to avoid copy_to_user in a tight loop by writing directly to the page). However, this is rarely worth it unless you're doing something complex – copy_to_user is heavily optimized (uses vectorized instructions, etc.). Direct access via pinned pages could be useful if you want to perform in-place modification of user buffer in kernel (maybe filtering an image in place? but doing so crosses user-kernel boundary anyway). Also, if you do use GUP for CPU access, you must still be careful about cache coherence (though if CPU only, not an issue, that's for DMA really) and memory consistency (the user could concurrently modify memory unless you have some locking). Generally,

copy_to_user is simpler and safer for CPU copy. Use GUP when you need to interface with hardware or get zero-copy with asynchronous operations.

Releasing pages and concurrency: When you call `put_page`, it decrements refcount. If that page was also mapped in userspace (it is), the user mapping still holds one reference (actually the process's page table mapping counts as an "implicit" reference until page is freed). If the user process exits or unmaps while you have it pinned, the page won't actually be freed until you unpin. So pinning ensures safe against user freeing. Conversely, if you forget to unpin, the page will never be freed until reboot or module unload (memory leak). Also, pinning a page doesn't prevent the *user* from writing to it concurrently. If that's an issue (say you're computing something on the buffer), you might need to synchronize with user space (perhaps via user-facing semaphore or make the ioctl synchronous).

`pin_user_pages()` vs `get_user_pages()` in modern kernels (Advanced)

Starting around Linux 5.5, a distinction was made between long-term DMA pins and short-term access. Long-term DMA (RDMA, GPU userptr, V4L2 buffers, etc.) can cause issues with file-backed pages (breaking COW, etc.). So new APIs `pin_user_pages()` and `unpin_user_page()` were introduced [39] [40]. These are essentially the same as GUP, but they set a FOLL_PIN flag internally to mark the pages as pinned for DMA. This allows kernel to track these pages separately (via a pin count bias). The rule is: **use `pin_user_pages` for pages that will be DMA'd (especially long term), use `get_user_pages` for short-lived access that is purely CPU or doesn't need the new semantics** [40] [41]. In many cases, short-term DMA (like our immediate I/O example above) is fine with `get_user_pages`, but best practice is shifting to `pin_user_pages` for any DMA usage. The difference is mostly internal – as a driver author, you just need to pair `pin_user_pages` with `unpin_user_page` (or a loop of them) instead of `put_page`. Unpin internally does some extra accounting if needed. For our discussion, we won't delve deeper, but it's good to be aware if you see warnings about "DMA pages pinned" in logs – using the proper API helps avoid those.

Asynchronous I/O with get_user_pages: If you want truly asynchronous operations (not blocking the process while hardware DMA happens), you have to do something like above in an ioctl or `read` handler but in a non-blocking way. Typically you'd pin pages, submit DMA, and *return* immediately, later notifying the user when done (maybe via poll or another ioctl or a signal). This leads us into asynchronous I/O topic.

Asynchronous I/O (AIO) in Drivers

Asynchronous I/O allows a user process to initiate an I/O operation and not wait for its completion – the operation completes in the background and the process is notified (via event or callback) later. In kernel 2.6, the **AIO subsystem** was introduced for file descriptors: it provided new file operations `aio_read`, `aio_write`, etc., and an interface for user to retrieve completions. It was not widely used for char drivers (most didn't implement it) [42] [43], but understanding it is useful for high-performance drivers (like tape drives as LDD3 notes [44] or perhaps custom device doing overlapped I/O).

The Linux AIO interface (lio_listio, io_submit) – brief overview

User-space typically uses the POSIX AIO `lio_listio()` or `aio_read` functions, or Linux-specific io_submit for native AIO. For drivers, supporting AIO means implementing the file operations:

```
ssize_t (*aio_read)(struct kiocb *iocb, char __user *buf, size_t count, loff_t
offset);
ssize_t (*aio_write)(struct kiocb *iocb, const char __user *buf, size_t count,
loff_t offset);
```

(`aio_fsync` exists but mainly for filesystems). These are similar to regular `read`/`write` but take a `struct kiocb` (kernel I/O control block) instead of `struct file*` (though `kiocb->ki_filp` is the file). The `offset` is passed by value (since AIO doesn't use the file's current position – multiple ops can be pending). The driver's `aio_read` should initiate the I/O and return quickly. If the operation can complete immediately, it can do so and return the byte count (just like a normal read). If it will complete later, then the driver should **not** copy data to user or block; instead, it queues the operation internally (perhaps storing the `kiocb` pointer and request info). It then returns `-EIOCBQUEUED` to the kernel [45]. This special error code tells the kernel "I've got this, it's in progress." The kernel will not mark it as an error to the user; instead user will be notified via the AIO ring or event mechanism when it's done.

Later, when the data is ready or the operation finished (say your device got an interrupt), the driver must call `aio_complete(iocb, bytes, 0)` [46]. This function tells the AIO core that the `iocb` is complete, with `bytes` result (or negative error). The AIO library in user space will then know this I/O is done (for instance, `io_getevents` will return it).

During this, you might need to access the user buffer outside process context (e.g., in an interrupt handler). Because once you returned from `aio_read`, the process could be doing other things or even exit (though exit will wait for AIO to complete usually). **Important:** if you plan to copy to/from the user buffer later (after `aio_read` returned), you cannot call `copy_to_user` at that time because you won't be in user context and the user pages might not even be present or still there. So you **must pin the user pages** during the `aio_read` call (when you still have the process context) [47]. This is explicitly mentioned: the driver should "remember everything it needs about the operation, and return -EIOCBQUEUED" [47]. Remembering the operation info includes "arranging access to the user-space buffer; once you return, you will not have the opportunity to access that buffer while running in the context of the calling process" [48]. That typically means using `get_user_pages` to pin and get an kernel-mapped address or DMA mapping for it. Alternatively, you could copy the data into a kernel buffer right away (but that defeats the purpose of async if it's large).

Therefore, AIO almost always goes hand-in-hand with direct I/O (zero-copy) if performance is the goal. The driver likely uses DMA to the pinned user pages or at least a persistent kernel mapping of them (like vmap). If the driver were to just copy synchronously into an intermediate buffer, it might as well do a normal read.

Completion and sync: The AIO mechanism in 2.6 had some quirks. For example, the kernel can issue "synchronous IOCB" calls sometimes where it expects you to actually do it synchronously (due to resource limits). There's a `is_sync_kiocb(iocb)` to check if the IOCB must be handled synchronously [49]. If so, you should just perform as a normal read and return the result directly, rather than queue.

If implementing AIO fully is too much, remember that drivers can also rely on **non-blocking I/O and** `select`**/**`poll` to achieve async behavior. Many char drivers simply implement `read` in a non-blocking fashion: if no data, return -EAGAIN, and user uses `poll()` to wait for data ready. Or driver uses

`schedule_work` or similar to handle lengthy operations while returning -EAGAIN or something and notify via poll when done. This is a simpler form of async I/O, albeit not the POSIX AIO interface.

Modern async: io_uring (FYI)

Linux 5.1 introduced **io_uring**, a new async I/O interface that is more efficient and general than the older AIO. It allows system calls like read, write, send, etc., to be completed asynchronously using submission and completion rings. From a driver perspective, you typically do not have to implement anything special for io_uring; if your driver's read/write can return -EAGAIN (for non-blocking) and supports polling, the io_uring infrastructure can handle it by using your `poll` to know when to retry, etc. If your driver is purely synchronous always, io_uring can still use a helper thread to do it (so not truly async). The advantage of io_uring is that it doesn't require the driver to implement separate AIO callbacks. Many developers prefer it over the older AIO interface. So, a tip for driver writers: if you want to support async I/O in modern style, implement your operations in a non-blocking manner (when possible) and provide a `.poll` to signal read/write readiness. Then io_uring will "just work" with your driver, handling async submission in user space or via polling.

Wrap up: AIO is an advanced topic – for a beginner, it's okay to skip implementing it. The default is that the kernel will emulate AIO by creating threads (this is how `libaio` works for files that don't support real AIO). But we covered it because it ties into memory mapping: to do AIO properly, you often need to pin user memory (direct I/O) so hardware or the kernel can access it out-of-context.

Now, with memory mapping and direct user memory access covered, let's move to the final and perhaps most important topic: **Direct Memory Access (DMA)**.

Direct Memory Access (DMA) – Theory and Practice

"DMA is the hardware mechanism that allows peripheral components to transfer their I/O data directly to and from main memory without the need to involve the system processor" [50]. Instead of the CPU executing loads and stores to move data between device and memory, the device (or a DMA controller) takes over the bus and transfers data autonomously. This frees the CPU and can be much faster, especially for large data transfers.

Nearly all high-speed device drivers (disk controllers, network cards, graphics, etc.) use DMA. In driver code, using DMA involves a set of steps to prepare memory, initiate the transfer, and clean up when done. The Linux kernel provides the **DMA API** to abstract device differences and handle cache coherence, addressing limitations, and so on. We'll discuss the important aspects:

- Coherent vs Streaming DMA mappings (two ways to use DMA buffers).
- Allocating DMA-able memory with `dma_alloc_coherent` or streaming APIs.
- The need for cache management and how the DMA API handles it.
- Scatter-Gather DMA and the scatterlist helper functions.
- Dealing with device addressing limitations (ISA DMA, 32-bit devices on >4GB RAM, etc.).
- DMA pools for small consistent buffers.
- Steps to perform a DMA transfer (from driver's perspective).

The DMA Address Space and Mapping

Recall the concept of **bus addresses** from the memory model section. A device doesn't see CPU virtual addresses; it issues addresses on the bus (PCI, etc.) which might map to physical memory through possibly an IOMMU. The kernel's DMA API revolves around providing a device with the correct address to use for DMA and ensuring the memory is prepared (in terms of caching and permissions).

Think of it like this: as a driver, you have a CPU virtual address of a buffer. To get the device to access it, you must:

1. Make sure the buffer resides in memory the device *can* access (e.g., not on high memory if device lacks that range, or not swapped out).
2. Obtain the **DMA (bus) address** corresponding to that buffer.
3. If the CPU cache could interfere (on non-coherent arch), perform cache flush or invalidation so device and CPU see consistent data.
4. Tell the device to do DMA at that address.
5. After DMA, synchronize caches again if needed (for CPU to see fresh data or device to not get stale data next time).

The DMA API functions take care of steps 1, 2, 3, and 5 in a portable way. For example, on x86 (which is cache-coherent), the cache management steps might be no-ops; on ARM, they might do actual flush instructions.

DMA Mapping Types: Coherent vs Streaming

Linux defines two main ways a driver can use memory for DMA:

- **Consistent (Coherent) DMA:** Memory that is set aside for device access and is guaranteed to be coherent for both device and CPU at all times. Both can access it without explicit cache flushes each time. Typically, this is achieved by mapping the memory as uncached (or write-combining) on non-coherent platforms, or by using a region of memory that is snoopable by the hardware. You obtain such memory by `dma_alloc_coherent()` [51]. The API returns a CPU address and a DMA address for the buffer [51]. Because it's uncached (on systems where needed), the CPU should not expect normal caching performance – but coherency is assured. Coherent DMA is often used for small control data or circular buffers where both sides might touch the data frequently (e.g., a network card's descriptor ring). It's less common for large data payloads due to performance (uncached large data is slow for CPU) and memory fragmentation (these buffers must be contiguous physical memory).

- **Streaming DMA:** This is a more flexible approach for dynamic data buffers. You use normal memory (like from `kmalloc` or even stack or user pages) for DMA, but immediately before the DMA transfer you **map** the buffer for the device (`dma_map_single` or `dma_map_page`) [52], and after the transfer you **unmap** it (`dma_unmap_single`) [53]. The mapping functions will handle cache flush/invalidates as needed and will return a `dma_addr_t` (bus address) for the device [54]. Between the map and unmap, you should treat the buffer as owned by the device – the CPU should not modify or read it (for DMA_FROM_DEVICE, CPU should not read stale data before device writes; for DMA_TO_DEVICE, CPU should not modify it after giving to device, etc.). After unmapping, the CPU

regains ownership and can safely access the data (with caches updated appropriately by the unmap call on platforms that need it). Streaming mapping is appropriate for data buffers that are used once or intermittently, especially large buffers or when you don't want to keep a permanent DMA buffer around. It's also the only way to DMA to user memory or other arbitrary memory not allocated by `dma_alloc_coherent`.

Analogy: If coherent DMA is like having a shared memory area always visible to both parties (but possibly slower for one party due to no caching), streaming DMA is like checking out a piece of memory to the device (ensuring it's clean), then later checking it back in to the CPU (ensuring changes are reflected).

Using `dma_alloc_coherent` (Coherent memory allocation)

To allocate a consistent DMA buffer, do:

```
dma_addr_t dma_handle;
cpu_addr = dma_alloc_coherent(dev, size, &dma_handle, GFP_KERNEL);
if (!cpu_addr) {
    // handle allocation failure
}
```

Here, `dev` is the struct device for your hardware (used to find the right bus/IOMMU info). `size` is the length. `dma_handle` will be filled with the DMA (bus) address to give the device [51]. The returned `cpu_addr` is a kernel virtual address you can use to access the buffer. This memory is *not* swappable and is page-aligned. On some platforms it might come from a special DMA zone or have certain alignment padding.

When done with it (like at driver unload or if you only needed it temporarily), free it with:

```
dma_free_coherent(dev, size, cpu_addr, dma_handle);
```

Coherent memory has some rules: it might be at least one page in size (some implementations round up allocations to whole pages or more) [55]. If you request a very small size, you might still get a whole page (and `dma_free_coherent` expects the same size). If your device needs many small DMA buffers, allocating a ton of full pages is wasteful – that's what DMA pools are for (later).

Also, coherent memory is uncached on many architectures. This means if your CPU tries to read/write it, it's going straight to RAM or device, which is slower than cache. So you wouldn't want to do heavy computations on data in a coherent buffer – better to copy it to a normal buffer first if needed for CPU processing.

Where is it allocated? On x86, `dma_alloc_coherent` usually comes from normal memory (because x86 is cache-coherent, it doesn't have to do uncached, though it might set PAT to WC or UC as requested). On ARM without cache coherence, it could come from a separately designated region or use noncacheable

mappings. You don't usually care; just remember not to alias this memory with normal cached mappings (the API ensures that).

Using Streaming DMA mappings (`dma_map_single`, `dma_map_page`)

When you have a buffer from general allocation and want to DMA it, use `dma_map_single(dev, cpu_addr, size, direction)` [52]. Directions are `DMA_TO_DEVICE` (you're sending data out, so the device will read from this buffer), `DMA_FROM_DEVICE` (device will write into this buffer), or `DMA_BIDIRECTIONAL` (device may both read and write, e.g., bidirectional ring buffer). The function returns a `dma_addr_t` (the bus address) [54]. You then program your device with that address.

After the DMA completes, call `dma_unmap_single(dev, dma_addr, size, direction)` to release the mapping [53]. For devices that use streaming DMA, you must do this unmap; you can't reuse the address across multiple operations without re-mapping (unless you deliberately keep it mapped, which is not typical and not advisable because of cache effects and potential mapping leaks – if you do, you might need `dma_sync_single_for_cpu/device` calls to synchronize without unmapping, but that's more advanced).

What does map do internally? Possibly several things: If an IOMMU is present, it creates an IOMMU mapping (so the `dma_addr_t` might be an iova (IO virtual address) that the device should use). If the device can't access the full physical memory and the buffer is above its range, the kernel might use a **bounce buffer** (copy data to a low memory buffer and map that for device). It will also flush caches if `direction` is DMA_TO_DEVICE (to push any dirty CPU cache lines out to memory so device sees current data), or invalidate cache if `FROM_DEVICE` (so stale CPU lines won't overwrite data that device writes) [56]. On a coherent platform, these flushes might be no-op because hardware does cache snoop (like on x86, if a line is in CPU cache and dirty, a DMA read will either cause a snoop or the map function will use CLWB/MFENCE or something – but essentially you don't have to worry).

Checking mapping errors: Always check `dma_mapping_error(dev, dma_addr)` after mapping [57]. If it returns nonzero, the mapping failed (perhaps out of IOMMU space or bounce buffer). Handle that (e.g., free some buffers or panic if critical). Usually mapping failures are rare if memory was gotten with GFP_DMA for restrictions or if using an IOMMU (which can still fail under heavy load).

One-time setup vs repeated mapping: If you have a long-lived buffer you use for DMA repeatedly (like a persistent kernel buffer for a device), you might wonder if you should allocate it with `dma_alloc_coherent` or just `kmalloc` and map on the fly. If performance is critical and the buffer is frequently used, `dma_alloc_coherent` is simpler (no repeated map/unmap and no cache flush overhead each time, at cost of uncached access possibly). If the usage is infrequent or buffer is large and you want it cached when CPU processes it (like a big image that the CPU will manipulate after DMA), then maybe allocate normally and use streaming mapping so that after DMA, once unmap, the data is back in cache (or can be loaded to cache normally). Keep in mind, after `dma_unmap_single(DMA_FROM_DEVICE)`, on noncoherent arch, the API will typically invalidate or clean the cache such that CPU sees device's writes. On coherent arch, it might do nothing, since coherence was automatic.

DMA Scatter-Gather (SG) and `dma_map_sg`

Scatter-gather refers to the device's ability to perform DMA to/from a list of memory segments in one logical transfer. Instead of one contiguous buffer, the data can be scattered in memory, and the device is given a list (often called DMA descriptors) of (address, length) pairs to process sequentially. Many advanced devices support SG (all SCSI, SATA, NVMe, network cards for jumbo frames, etc.). Linux's block layer uses scatterlists heavily to gather page cache pages for disk I/O, for example.

The kernel uses the `struct scatterlist` to represent a list of memory chunks. When you have an array of pages from `get_user_pages` or an array of kernel buffers, you can set up a scatterlist (with `sg_init_table` and then for each entry `sg_set_page` or `sg_set_buf`). Then you call `dma_map_sg(dev, sglist, nents, direction)`. This will map all entries for DMA, possibly merging adjacent ones if the hardware allows contiguous mapping, and return the number of SG entries in the DMA address space [58]. After mapping, each `sg->dma_address` and `sg->dma_length` are set for use by the device (or you might pass the sglist to an API if using a kernel subsytem). After DMA, call `dma_unmap_sg(dev, sglist, nents, direction)`. The merging means the number of entries the device has to handle could be <= original number of pages.

For example, if you have 4 physically contiguous pages in a scatterlist, `dma_map_sg` might coalesce them into one entry (sg[0].dma_length = 4*PAGE_SIZE) and return 1. If they were separate, it returns maybe 4, with each entry filled.

Scatterlist Example: If implementing a block device driver, the request comes with a bio of segments or already an SG list of pages. The driver would map that sg, get the DMA addresses, then program the device's SG DMA engine with those addresses. Many bus mastering drivers in Linux do exactly this.

For char drivers, you might not often need `dma_map_sg` unless you are doing something like the earlier example of directly DMAing into user pages – there, if the buffer spanned multiple pages, using an SG mapping is convenient. Indeed, the kernel's own implementation of O_DIRECT for file I/O does this: get_user_pages, then dma_map_sg on those pages [36]. (If hardware doesn't support SG, the block layer would have already merged into a contiguous bounce buffer, but that's done at higher layers.)

Ensuring Device Addressability – DMA Masks and Bounce Buffers

Not all devices can address all physical memory. For instance, a 32-bit PCI device can only address the lower 4GB of memory (DMA addresses must fit in 32 bits). If a system has more RAM (say 16GB), and the kernel allocates a buffer above 4GB, that device cannot reach it directly. The kernel's DMA subsystem manages this via *DMA masks*. Each device has a DMA mask (typically 32-bit by default for PCI devices, 24-bit for truly old ones, etc., and many modern 64-bit capable devices will set a 64-bit mask).

As a driver, you should set the appropriate mask in probe: `dma_set_mask(&pdev->dev, DMA_BIT_MASK(n))` where n is the number of address bits your device supports. For example, if your device can do 64-bit DMA (DAC), you'd attempt `dma_set_mask(dev, DMA_BIT_MASK(64))`. If that fails (perhaps no 64-bit addressing available or IOMMU not support that), you might fallback to 32: `dma_set_mask(dev, DMA_BIT_MASK(32))` [59] [60]. Also set `dma_set_coherent_mask` for coherent allocations similarly. Many drivers just call `dma_set_mask` and if it fails, print an error and bail out if they

really need high memory; others can work with 32-bit (the failure means they'll have to bounce if needed, which the DMA API will do via swiotlb on x86 typically).

If a device has a 32-bit mask, the `dma_alloc_coherent` will try to ensure memory comes from the first 4GB (often it does anyway). `dma_map_single` will, if it encounters a buffer above 4GB, use a bounce buffer (if SWIOTLB is enabled) – basically it allocates a buffer in low memory and copies data there, then gives device that address [58] [61]. After DMA, it copies back. This is transparent to the driver, except performance is lower. But it prevents outright failure in many cases. On some systems, an IOMMU might map the high address into a low bus address, also transparent.

ISA DMA (16MB limit): ISA devices (old sound cards, floppies, etc., using the 8237 DMA controller) have a 24-bit address limit (16MB). Linux historically defined `ZONE_DMA` for the first 16MB, and you use `GFP_DMA` to allocate memory there [22]. The DMA API also knows the mask (16MB ~ 24-bit) and will bounce if needed. But usually for ISA, drivers explicitly use `GFP_DMA`. For example, the SoundBlaster 16 driver would allocate its audio buffer with `snd_dma_alloc_pages_fallback(SNDRV_DMA_TYPE_DEV, snd_dma_isa_buf, ...)` which eventually ensures low mem, etc. If you ever write a driver for ISA, yes, use GFP_DMA.

64-bit PCI (DAC): If your device supports 64-bit, just set mask to 64. On 64-bit platforms, dma_set_mask(64) should succeed and then you don't worry about bouncing. On 32-bit platform with more than 4GB RAM, a 64-bit capable device could in theory DMA above 4GB, but 32-bit CPU cannot easily have >4GB physical memory without PAE. In PAE, addresses are 36-bit physical and device could use DAC to reach up there. This did happen in some servers. But nowadays, mostly moot since most are 64-bit.

Ensuring Cache Coherency and Memory Barriers (Advanced)

One subtlety: even with DMA APIs doing flushes, you must ensure proper memory ordering. If the CPU writes to a DMA buffer and then immediately tells device "go!", you might need a **write memory barrier** (wmb()) before writing to device registers to ensure the CPU's writes to memory are visible (on some weakly ordered arch). The DMA API's mapping might include such barriers, but if you're reusing an already mapped coherent buffer, you must handle it. Similarly, after device writes to memory and triggers interrupt, the interrupt handler might need to do a **read memory barrier** (rmb()) before reading the data to ensure it's not seeing stale reads. On x86, these are usually no-ops or automatically handled by cache snooping. On some, they matter. The details are beyond our scope here, but be aware of the concept of memory barriers in DMA interactions.

DMA Pools for Small Buffers

If you need many small (sub-page) DMA buffers, calling `dma_alloc_coherent` for each (which might give a full page) wastes memory. Instead, use **DMA pools** [55] [62]. A DMA pool lets you allocate chunks of a page for DMA in a safe way (taking care of boundary alignment and such).

Usage:

```
struct dma_pool *pool = dma_pool_create("mypool", dev, size, align,
allocation_increment);
```

`size` is the element size you need (e.g., 64 bytes for some hardware descriptor), `align` is required alignment (maybe 64 as well), `allocation_increment` can often be 0 or a power of two indicating how allocations are grouped, if needed.

Then get memory:

```
void *cpu_ptr = dma_pool_alloc(pool, GFP_KERNEL, &dma_handle);
```

Free memory with `dma_pool_free(pool, cpu_ptr, dma_handle)`. Finally `dma_pool_destroy(pool)` when done (like at module exit).

Internally, the pool will allocate whole pages with `dma_alloc_coherent` and parcel them out, keeping track of which parts are free. It ensures that the memory for each chunk is DMA-coherent and appropriately aligned, etc. It's commonly used in drivers for things like ring descriptors or USB transfer descriptors. For example, USB controllers often need 32-byte aligned descriptors; the USB core uses dma_pool for those.

If each `dma_alloc_coherent` has a minimum one-page, and you need 100 buffers of 64 bytes, pool will allocate maybe 25 pages and serve 4 per page, instead of 100 pages.

ISA DMA and the 8237 Controller (Legacy)

(Legacy content, but explained for completeness) The ISA bus on PCs (and similar buses on old systems) had a separate DMA controller chip (the Intel 8237 in the IBM PC). Devices like floppy disk controllers, old sound cards, parallel port in ECP mode, etc., would use this controller to do DMA. The 8237 had severe limitations: it could only address 16MB (24-bit addresses), only certain channels could do 16-bit transfers vs 8-bit, and it couldn't scatter-gather – it just did single contiguous transfers. Also, it had a small number of channels (8 channels, some reserved by system like channel 4 used internally as cascade).

In Linux, there's an API to use ISA DMA:

- `request_dma(channel, "name")` – like requesting an IRQ, you request exclusive use of a DMA channel [63].
- `free_dma(channel)` – release it when done [63].

You should do `request_dma` at open() and `free_dma` at release() of your device (if it's a multiplexed resource) [64]. If you hold it constantly (maybe a built-in device), then request at driver init.

To start an ISA DMA transfer, drivers use functions defined in `<asm/dma.h>` on i386 (or Linux DMA API provides some in drivers/dma/isa-dma.c):

- `set_dma_mode(channel, mode)` – mode is read or write (from memory perspective).

- `set_dma_addr(channel, dma_addr)` – the low 24 bits of address where transfer starts [65]. Because ISA DMA can only do 24-bit, if your buffer is above 16MB, it won't work (hence GFP_DMA to ensure below 16MB).
- `set_dma_count(channel, count)` – number of bytes (up to 64k for 8-bit channels, 128k for 16-bit combined channels).

Then `enable_dma(channel)` to start, etc. When device triggers, the 8237 will transfer and maybe generate an IRQ when done (depends on device).

There are also `disable_dma`, `clear_dma_ff` (flip-flop) and others. The details aren't crucial unless writing a driver for such hardware. The key point: **if you find yourself writing an ISA device driver**, you likely need to follow these steps and know the limitations. The LDD3 text covered this and warned to be careful with the shared DMA controller (use locking if you manipulate its registers, although Linux's isa-dma API might handle locking via `claim_dma_lock()`) [66] [67].

Nowadays, ISA DMA is mostly of historical interest. The Raspberry Pi certainly does not have an 8237 – it has its own internal DMA controllers which operate differently (and Linux treats those as general-purpose DMA engines accessible via `dmaengine` API if at all).

So, summarizing:

- **Zone DMA (first 16MB)** is there for legacy ISA. Use GFP_DMA if needed.
- **API**: `request_dma` / `free_dma` to claim channels, and `set_dma_addr/count/mode` to program.
- It's a shared resource (only one device can use a channel at a time).
- The driver must coordinate with the device's registers as well: often you program the device to start a DMA read or write and simultaneously setup the 8237 to match.

We include this because older texts do, but in modern driver writing, you might never need this unless dealing with retro hardware or PC/104 boards.

Summary of Best Practices (Memory Mapping & DMA)

We've covered a lot. Let's boil down some best practices:

- **Use the DMA API for anything related to DMA.** Avoid older functions like `virt_to_bus` or manual cache flush instructions – the DMA API handles IOMMUs, bounce buffers, and cache coherency for you [68] [24].

- **Choose the right mapping method:** If you have device memory or reserved buffers, `remap_pfn_range` can map it to user space (set proper flags). If you have pageable memory or want on-demand allocation, use a fault handler and `vm_insert_page`. Don't expose normal kernel memory to user space without marking it IO or reserved appropriately.

- **Pin user pages for direct IO**: If transferring directly between device and user memory, pin the pages (`get_user_pages` / `pin_user_pages`), use `dma_map_page` on them for the device. And always unpin (put_page) and sync after I/O [2] [69].

- **Keep track of who owns a DMA buffer at any time:** If the device is writing to it, the CPU should not. Use memory barriers if needed when handing off and completing transfers.

- **Don't reinvent sync routines:** Use provided calls like `dma_sync_single_for_cpu` if you need to access a still-mapped streaming DMA buffer from the CPU (rare case), or `dma_sync_single_for_device` if CPU touched a coherent buffer and you want to ensure device sees updated data before DMA.

- **Use consistent memory for consistent data (small control structures)** and streaming for bulk data.

- **Set DMA mask** early in your driver (before allocating DMA memory). The kernel might default to 32-bit but it's good to be explicit, especially if >4GB addressing or if on some architecture with weird limitations.

- **Check return values** – whether from `get_user_pages` (which might fail if memory is invalid or locked), or `dma_map_single` (which might fail if out of mapping resources) [70]. Handle errors gracefully (perhaps by falling back to simpler method or returning error to user).

- **Manage lifetimes:** If user does mmap and your driver is unloaded or device removed, beware – the memory is still mapped in user space. Typically, on device removal, you'd need to zap or remap those pages to safe placeholders. The kernel can help: setting `VM_IO` and `VM_PFNMAP` prevents those from being dumped or swapped, but if device is gone, accesses might fault/crash. Some drivers revoke mappings on shutdown via global flags. This is an advanced scenario but mention that if your hardware can vanish (hotpluggable) and you allow mmap, you need a strategy to invalidate mappings on removal (often not trivial – many drivers just disallow unbinding the driver while mmap active or rely on user to close).

Finally, we proceed to a practical project that puts several of these ideas together.

Project: DMA Buffer Sharing with mmap – Raspberry Pi Char Driver

As a hands-on exercise, let's outline a simple project for a 64-bit Raspberry Pi (or any ARM64 board) that demonstrates memory mapping and DMA concepts. We'll create a **character driver** that:

- Allocates a contiguous DMA-capable buffer in the kernel.
- Exposes this buffer to user space via `mmap`.
- Uses a simulated DMA operation to copy data into this buffer (in a real scenario, this could be done by hardware; we'll emulate it with kernel code for learning).
- Allows the user application to be notified when data is ready (using `poll` / `select` or blocking read).

Goal: The user program will `mmap` the buffer and then trigger a "DMA" operation. The driver will asynchronously fill the buffer (simulate device DMA) and alert the user. The user can then directly read the new data from the memory region. This shows zero-copy transfer (no extra kernel-user copy) and use of DMA-safe memory.

Step 1: Driver Setup and Buffer Allocation

In the driver's init (or on open), allocate a DMA buffer. We can use `dma_alloc_coherent` for simplicity, size say 4096 bytes (one page). Alternatively, use `kmalloc` with GFP_DMA or CMA – but `dma_alloc_coherent` is straightforward and ensures alignment and uncached memory on Pi (Pi's ARM core is not hardware cache-coherent with some peripherals, so consistent memory will likely be noncached).

Store the `cpu_addr` and `dma_handle`. Also set up any synchronization primitives (a wait queue or completion to signal data ready).

Step 2: Implement mmap in the driver

Provide a `mydrv_mmap` that maps the allocated buffer to user. We have the physical address (or rather, DMA handle which on ARM without IOMMU is the physical address). Use `remap_pfn_range`: `pfn = dma_handle >> PAGE_SHIFT`. Mark VMA as IO and don't dump. Since `dma_alloc_coherent` gave us coherent memory, it's probably mapped as device memory (strongly ordered, no caching). We should still do `pgprot_noncached` on VMA to match (or `pgprot_writecombine` if appropriate). For Pi's RAM, it might allow a stronger caching if the hardware is coherent; but Pi's internal DMA is not snooping the L1/L2, so noncached is safest. The driver should also set `vma->vm_private_data = pointer to buffer` if needed for reference, but here we have global.

Step 3: Expose a way to trigger "DMA"

We can implement a `write` or `ioctl` that when called, the driver will simulate a DMA operation. For example, user does `write(fd, "GO", 2)` or an `ioctl(fd, START_DMA)`. The driver upon receiving this will start a kernel timer or workqueue that waits a bit (simulate DMA latency) then writes some pattern or data into the DMA buffer. After writing, it will wake up the user's process or mark data ready.

Alternatively, implement `read` in blocking mode: On `read()`, driver waits, then when data "arrives" (triggered internally or by an earlier action), it unblocks. But since we are using mmap, we don't actually need to copy any data to user in read; read could just block until buffer updated then return number of bytes (without actually transferring, because user can already see the data in mmap). Or we skip read entirely and have user just do busy-wait or poll on the memory. But better is to use `poll()`: implement `mydrv_poll` to wait for data-ready event (driven by our DMA simulation).

So, let's do this: The driver has a waitqueue `waitq`. When simulation finishes filling buffer, it does `wake_up_interruptible(&waitq)`. The `poll` function will check a flag (e.g., `data_ready` flag protected by lock or atomic) and if ready, return POLLIN|POLLRDNORM; if not, add the waitqueue to poll and return 0. The user can then do `poll()` or `select()` on the fd to wait.

Alternatively, simpler, the driver's `read` can sleep on waitqueue until data_ready then return number of bytes. But since data is already in the mmap buffer, the read doesn't need to copy it – it could just consume the event. In fact, we might just use read as the trigger: user calls read(), driver blocks until DMA done, then instead of copying to user (which we want to avoid), it just returns the size indicating "data available in mmap buffer". This is a bit unusual for an API but okay for a test. Or the read could return some small token or even 0 to indicate "data done".

To keep it simple: use an ioctl or write to *start* the DMA, and use read to *wait for completion*. For instance, user does ioctl(START), then does a read() which blocks until complete and then returns number of bytes transferred (and maybe resets the flag).

This way, actual data transfer is via memory mapping (no copy). The read is only used as a synchronization point to know when new data is ready.

Step 4: Simulate the DMA in kernel

We can use a kernel delayed work or tasklet, or even the hrtimer. But easiest: when user triggers, spawn a kernel thread or schedule a work that sleeps for, say, 100ms (simulate device delay) then writes to the buffer. The content could be anything: maybe fill with a pattern or incrementing numbers to see changes.

After writing, wake up the waitqueue and set data_ready = true. The user's blocked read unblocks and returns.

If using `poll`, we ensure to call `wake_up_interruptible(&waitq)` which will notify poll too.

Step 5: User-space program

Write a test C program that opens the device, mmaps the buffer, then perhaps writes some known pattern into it (or zeros it), then triggers the ioctl for DMA. Then either do a blocking read or use poll to wait. Once unblocked, print the buffer contents (which the kernel filled). This demonstrates that the user saw data appear without any read copying.

We must ensure the buffer is cache-coherent. Since we used `dma_alloc_coherent`, on ARM that buffer is non-cached; the user-mapped view of it in userspace will also be noncached (since we used pgprot_noncached). That means from user's perspective, normal loads and stores go directly to memory (which is fine, just maybe slower – but 4k size we won't notice). If we didn't do that and the user mapping was cached, the user might not see the new data because it could be stuck in cache (if CPU caching were allowed on that region, which we prevented via pgprot_noncached).

Alternatively, if we wanted to be fancy, we could allocate normal cached memory and manually flush/invalidate caches on both sides – but that's too low-level. Coherent memory is simpler for this demonstration.

Step 6: Clean up

On driver unload or close, free the DMA buffer. If multiple processes might mmap, reference counting might be needed, but we can keep it simple (one user at a time or a static mapping).

On Raspberry Pi, you'll load this module and run the program as root (since /dev entry likely root unless we do udev rules). You should see the data.

Memory safety: Because the user can write to the mapped buffer too, perhaps the simulation could read something the user wrote (like treat it as a command or such). But we won't complicate that. Just one direction.

Potential issues: If the user does not call the read to clear the flag and starts another DMA, our logic should handle it (maybe we disallow if previous not read yet, or override). But for simplicity, perhaps block a second start until previous done.

The main point is to show usage of `dma_alloc_coherent` + `remap_pfn_range` + synchronization (which uses no additional copying).

Code Sketch (not full code, but enough):

```
#define BUF_SIZE 4096
static void *dma_buf;
static dma_addr_t dma_handle;
static DECLARE_WAIT_QUEUE_HEAD(waitq);
static atomic_t data_ready = ATOMIC_INIT(0);

static ssize_t mydrv_read(struct file *filp, char __user *ubuf, size_t count,
loff_t *offp) {
    // Only used to wait for data ready
    if (!atomic_read(&data_ready)) {
        if (filp->f_flags & O_NONBLOCK)
            return -EAGAIN;
        // Block until data_ready set
        int ret = wait_event_interruptible(waitq, atomic_read(&data_ready));
        if (ret)
            return -ERESTARTSYS;
    }
    // At this point, data_ready == 1
    size_t len = BUF_SIZE; // or some actual length of data if we simulate
partial
    atomic_set(&data_ready, 0);
    // Return length to indicate new data (we won't copy to user since it's in
mmap)
    return (len <= count) ? len : count;
}

static ssize_t mydrv_write(struct file *filp, const char __user *ubuf, size_t
count, loff_t *offp) {
    // Could use this to trigger DMA as well
    if (count > 0) {
        // For example, any write triggers the DMA simulation
        schedule_dma_work();
    }
```

```
    return count;
}

static long mydrv_ioctl(struct file *f, unsigned int cmd, unsigned long arg) {
    if (cmd == START_DMA_CMD) {
        schedule_dma_work();
        return 0;
    }
    return -ENOTTY;
}

static int mydrv_mmap(struct file *filp, struct vm_area_struct *vma) {
    if (vma->vm_end - vma->vm_start > BUF_SIZE)
        return -EINVAL;
    vma->vm_page_prot = pgprot_writecombine(vma->vm_page_prot);
    vma->vm_flags |= VM_IO | VM_DONTEXPAND | VM_DONTDUMP;
    unsigned long pfn = dma_handle >> PAGE_SHIFT;
    if (remap_pfn_range(vma, vma->vm_start, pfn, BUF_SIZE, vma->vm_page_prot))
        return -EAGAIN;
    return 0;
}
```

The `schedule_dma_work()` would be something like:

```
static void dma_work_handler(struct work_struct *work) {
    // Simulate DMA by filling buffer
    char *buf = (char*)dma_buf;
    for (int i = 0; i < BUF_SIZE; ++i)
        buf[i] = (char)(i & 0xFF); // or any pattern
    // maybe msleep(100); to simulate delay
    atomic_set(&data_ready, 1);
    wake_up_interruptible(&waitq);
}
```

We would set up this workqueue or use a delayed work in module init (e.g., INIT_WORK and a global work struct, and in schedule_dma_work just queue it).

Testing: The user app opens `/dev/mydma`, does `mmap` of one page, gets pointer. Then perhaps writes one byte to trigger or calls ioctl. Then does `read(fd, buf, BUF_SIZE)` (or smaller) to wait for completion. The read returns number of bytes. The user then prints first few bytes from the mapped memory to confirm they match what kernel put.

We'll likely see 0,1,2,...255 repeating as per our fill pattern.

This project touches on: allocation in low memory (though Pi's memory <4GB so no mask issue), mapping to user, cache coherency (ensured by coherent allocation), and user/kernel sync (via waitqueue).

Wrap-up: This tutorial covered the gamut from understanding addresses and kernel memory management, to implementing `mmap` for device drivers, to pinning user memory for direct I/O, to using the DMA API for high-performance data transfer. By updating the LDD3 content to Linux 6.x (removing deprecated interfaces like `.nopage` and using modern pin_user_pages and DMA APIs), we've equipped you to handle memory mapping and DMA in contemporary kernel development. The Raspberry Pi example ties it together in a practical way – try coding it and observing the results on real hardware for a full learning experience!

---

1 2 3 4 5 8 11 12 13 14 15 16 17 18 19 20 22 25 26 32 36 37 38 42 43 44 45 46 47 48 49 51 52 55 ch15.pdf
file://file_00000000d8a071f5804e97e9dad14761

6 7 21 31 35 50 58 59 60 61 62 63 64 65 66 67 kdlp.underground.software
https://kdlp.underground.software/slides/mm_dma.html

9 10 23 24 53 54 57 68 69 70 Dynamic DMA mapping Guide — The Linux Kernel documentation
https://docs.kernel.org/core-api/dma-api-howto.html

27 28 29 30 Memory Areas
https://litux.nl/mirror/kerneldevelopment/0672327201/ch14lev1sec2.html

33 Linux Kernel Memory Map Operations · Better Tomorrow with Computer Science
https://insujang.github.io/2017-04-07/linux-kernel-memory-map-operations/

34 remove nopage - LWN.net
https://lwn.net/Articles/258113/

39 40 41 pin_user_pages() and related calls — The Linux Kernel documentation
https://docs.kernel.org/core-api/pin_user_pages.html

56 Direct Memory Access (DMA) and Interrupt Handling
https://www.eventhelix.com/fault-handling/dma-interrupt-handling/