CHAPTER 15

Memory Mapping and DMA

This chapter delves into the area of Linux memory management, with an emphasis on techniques that are useful to the device driver writer. Many types of driver programming require some understanding of how the virtual memory subsystem works; the material we cover in this chapter comes in handy more than once as we get into some of the more complex and performance-critical subsystems. The virtual memory subsystem is also a highly interesting part of the core Linux kernel and, therefore, it merits a look.

The material in this chapter is divided into three sections:

- The first covers the implementation of the *mmap* system call, which allows the mapping of device memory directly into a user process's address space. Not all devices require *mmap* support, but, for some, mapping device memory can yield significant performance improvements.
- We then look at crossing the boundary from the other direction with a discussion of direct access to user-space pages. Relatively few drivers need this capability; in many cases, the kernel performs this sort of mapping without the driver even being aware of it. But an awareness of how to map user-space memory into the kernel (with *get_user_pages*) can be useful.
- The final section covers direct memory access (DMA) I/O operations, which provide peripherals with direct access to system memory.

Of course, all of these techniques require an understanding of how Linux memory management works, so we start with an overview of that subsystem.

Memory Management in Linux

Rather than describing the theory of memory management in operating systems, this section tries to pinpoint the main features of the Linux implementation. Although you do not need to be a Linux virtual memory guru to implement *mmap*, a basic overview of how things work is useful. What follows is a fairly lengthy description of

第15章

Chapter 15

内存映射和D MA

本章深入研究Linux内存管理领域，重点是对设备驱动程序作者有用的技术。许多类型的驱动程序程序需要了解虚拟内存子系统的工作方式。当我们进入一些更复杂和至关重要的子系统时，我们在本章中涵盖的材料不止一次。虚拟内存子系统也是核心Linux内核的一个非常有趣的部分，因此，它值得一看。

本章中的材料分为三个部分:

- 第一个涵盖了*mmap*系统调用的实现，该调用允许将设备内存直接映射到用户进程的地址空间中。并非所有设备都需要*mmap*支持，但是对于某些设备，映射设备存储器可以产生重大的性能改进。

- 然后，我们通过直接访问用户空间页面的方式来考虑从另一个方向越过边界。相对较少的驾驶员需要这种帽子。在许多情况下，内核可以执行此类映射，而无需驾驶员意识到它。但是，如何将用户空间内存映射到内核（使用*get_user_pages*）的意识很有用。

- 最后一部分涵盖了直接内存访问（DMA）I/O操作，该操作介绍了具有直接访问系统内存的外围设备。

当然，所有这些技术都需要了解Linux内存管理的工作原理，因此我们从该子系统的概述开始。

Linux中的内存管理

本节没有描述操作系统中的内存管理理论，而是试图查明Linux实现的主要特征。尽管您不需要成为Linux虚拟内存大师来实现*mmap*，但对工作方式的基本概述是有用的。接下来是一个相当漫长的描述

the data structures used by the kernel to manage memory. Once the necessary background has been covered, we can get into working with these structures.

Address Types

Linux is, of course, a virtual memory system, meaning that the addresses seen by user programs do not directly correspond to the physical addresses used by the hardware. Virtual memory introduces a layer of indirection that allows a number of nice things. With virtual memory, programs running on the system can allocate far more memory than is physically available; indeed, even a single process can have a virtual address space larger than the system's physical memory. Virtual memory also allows the program to play a number of tricks with the process's address space, including mapping the program's memory to device memory.

Thus far, we have talked about virtual and physical addresses, but a number of the details have been glossed over. The Linux system deals with several types of addresses, each with its own semantics. Unfortunately, the kernel code is not always very clear on exactly which type of address is being used in each situation, so the programmer must be careful.

The following is a list of address types used in Linux. Figure 15-1 shows how these address types relate to physical memory.

User virtual addresses
: These are the regular addresses seen by user-space programs. User addresses are either 32 or 64 bits in length, depending on the underlying hardware architecture, and each process has its own virtual address space.

Physical addresses
: The addresses used between the processor and the system's memory. Physical addresses are 32- or 64-bit quantities; even 32-bit systems can use larger physical addresses in some situations.

Bus addresses
: The addresses used between peripheral buses and memory. Often, they are the same as the physical addresses used by the processor, but that is not necessarily the case. Some architectures can provide an I/O memory management unit (IOMMU) that remaps addresses between a bus and main memory. An IOMMU can make life easier in a number of ways (making a buffer scattered in memory appear contiguous to the device, for example), but programming the IOMMU is an extra step that must be performed when setting up DMA operations. Bus addresses are highly architecture dependent, of course.

Kernel logical addresses
: These make up the normal address space of the kernel. These addresses map some portion (perhaps all) of main memory and are often treated as if they were physical addresses. On most architectures, logical addresses and their associated

内核使用的数据结构来管理内存。一旦涵盖了必要的背景，我们就可以使用这些结构。

地址类型

Linux当然是虚拟内存系统，这意味着用户程序所看到的地址与硬件使用的物理地址直接对应。虚拟内存引入了间接层，该层允许许多不错的东西。使用虚拟内存，系统上运行的程序可以分配比物理上可用的更多内存。确实，即使是一个过程也可以具有比系统物理内存大的虚拟地址空间。虚拟内存还允许程序在过程的地址空间中播放许多技巧，包括将程序的内存映射到设备内存。

到目前为止，我们已经谈论了虚拟和物理地址，但是许多细节已被掩盖。 Linux系统处理几种类型的地址，每个地址都有自己的语义。不幸的是，内核代码在每种情况下都确切使用哪种类型的地址，因此程序员必须小心。

以下是Linux中使用的地址类型的列表。图15-1显示了这些地址类型与物理内存的关系。

User virtual addresses
: 这些是用户空间程序看到的常规地址。用户地址的长度为32或64位，具体取决于基础硬件架构，每个过程都有其自己的虚拟地址空间。

Physical addresses
: 处理器和系统内存之间使用的地址。物理地址为32或64位；在某些情况下，即使是32位系统也可以使用较大的物理地址。

Bus addresses
: 外围总线和内存之间使用的地址。通常，它们与处理器使用的物理地址相同，但不一定是这种情况。某些体系结构可以提供一个I/O内存管理单元（IOMMU），该单元（IOMMU）重建总线和主内存之间的地址。 IOMMU可以通过多种方式使生活更轻松（例如，使散布在内存中的缓冲区与设备相接触），但是编程IMMU是在设置DMA操作时必须执行的额外步骤。当然，公交地址高度依赖建筑。

Kernel logical addresses
: 这些构成了内核的正常地址空间。这些地址绘制了一部分（也许是全部）主内存，通常被视为它们是物理地址。在大多数架构，逻辑地址及其相关的逻辑地址

physical addresses differ only by a constant offset. Logical addresses use the hardware's native pointer size and, therefore, may be unable to address all of physical memory on heavily equipped 32-bit systems. Logical addresses are usually stored in variables of type `unsigned long` or `void *`. Memory returned from *kmalloc* has a kernel logical address.

Kernel virtual addresses

Kernel virtual addresses are similar to logical addresses in that they are a mapping from a kernel-space address to a physical address. Kernel virtual addresses do not necessarily have the linear, one-to-one mapping to physical addresses that characterize the logical address space, however. All logical addresses *are* kernel virtual addresses, but many kernel virtual addresses are not logical addresses. For example, memory allocated by *vmalloc* has a virtual address (but no direct physical mapping). The *kmap* function (described later in this chapter) also returns virtual addresses. Virtual addresses are usually stored in pointer variables.

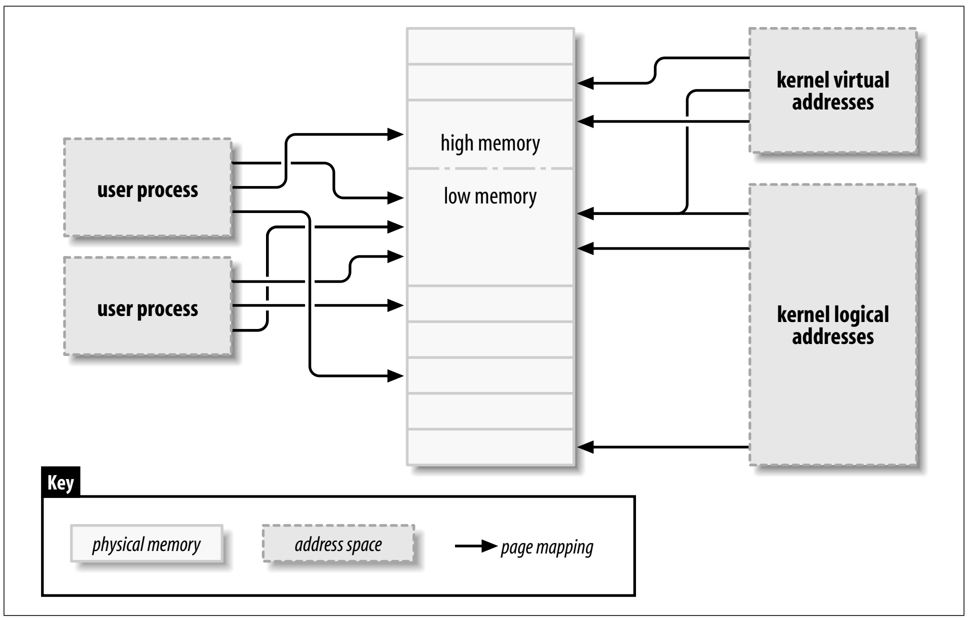

Figure 15-1. Address types used in Linux

If you have a logical address, the macro *__pa()* (defined in *<asm/page.h>*) returns its associated physical address. Physical addresses can be mapped back to logical addresses with *__va()*, but only for low-memory pages.

物理地址仅因恒定偏移而有所不同。逻辑地址使用硬件的本机指针尺寸，因此可能无法在装备精良的32位系统上解决所有物理内存。逻辑地址用通常存储在类型`unsigned long`或`void *`的变量中。从*kmalloc*返回的内存具有一个内核逻辑地址。

Kernel virtual addresses

内核虚拟地址类似于逻辑地址，因为它们是从内核空间地址到物理地址的地图。内核虚拟地址不一定具有线性的，一对一的映射到表征逻辑地址空间的物理地址。所有逻辑地址*are*内核虚拟地址，但是许多内核虚拟地址不是逻辑地址。例如，*vmalloc*分配的内存具有虚拟地址（但没有直接物理映射）。*kmap*函数（本章后面描述）还返回虚拟地址。虚拟地址通常存储在指针变量中。

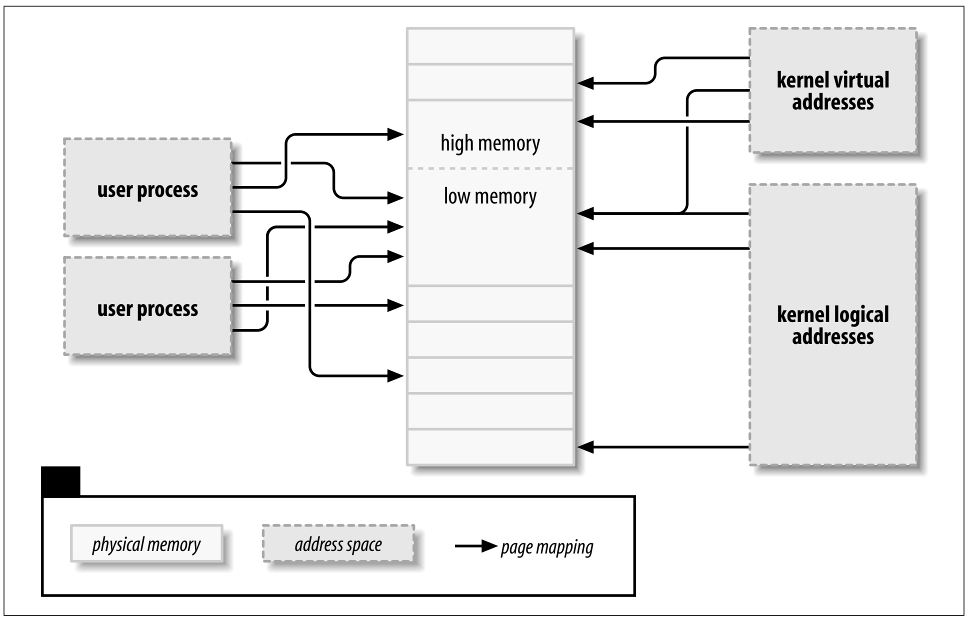

Figure 15-1. Address types used in Linux

如果您有一个逻辑地址，则<*asm/page.h*>)中定义的宏*__pa()* (将返回其关联的物理地址。物理地址可以用*__va()*映射回逻辑地址，但仅适用于低内存页面。

Different kernel functions require different types of addresses. It would be nice if there were different C types defined, so that the required address types were explicit, but we have no such luck. In this chapter, we try to be clear on which types of addresses are used where.

Physical Addresses and Pages

Physical memory is divided into discrete units called *pages*. Much of the system's internal handling of memory is done on a per-page basis. Page size varies from one architecture to the next, although most systems currently use 4096-byte pages. The constant `PAGE_SIZE` (defined in *<asm/page.h>*) gives the page size on any given architecture.

If you look at a memory address—virtual or physical—it is divisible into a page number and an offset within the page. If 4096-byte pages are being used, for example, the 12 least-significant bits are the offset, and the remaining, higher bits indicate the page number. If you discard the offset and shift the rest of an offset to the right, the result is called a *page frame number* (PFN). Shifting bits to convert between page frame numbers and addresses is a fairly common operation; the macro `PAGE_SHIFT` tells how many bits must be shifted to make this conversion.

High and Low Memory

The difference between logical and kernel virtual addresses is highlighted on 32-bit systems that are equipped with large amounts of memory. With 32 bits, it is possible to address 4 GB of memory. Linux on 32-bit systems has, until recently, been limited to substantially less memory than that, however, because of the way it sets up the virtual address space.

The kernel (on the x86 architecture, in the default configuration) splits the 4-GB virtual address space between user-space and the kernel; the same set of mappings is used in both contexts. A typical split dedicates 3 GB to user space, and 1 GB for kernel space.* The kernel's code and data structures must fit into that space, but the biggest consumer of kernel address space is virtual mappings for physical memory. The kernel cannot directly manipulate memory that is not mapped into the kernel's address space. The kernel, in other words, needs its own virtual address for any memory it must touch directly. Thus, for many years, the maximum amount of physical memory that could be handled by the kernel was the amount that could be mapped into the kernel's portion of the virtual address space, minus the space

* Many non-x86 architectures are able to efficiently do without the kernel/user-space split described here, so they can work with up to a 4-GB kernel address space on 32-bit systems. The constraints described in this section still apply to such systems when more than 4 GB of memory are installed, however.

不同的内核功能需要不同类型的地址。如果定义了不同的C类型，那就太好了，因此所需的地址类型是明确的，但是我们没有运气。在本章中，我们会尝试清楚使用哪些地址在哪里使用。

物理地址和页面

物理内存分为称为*pages*的离散单元。系统内部内部处理的大部分是每页完成的。页面大小从一个体系结构到另一个体系结构，尽管当前大多数系统都使用4096字节页面。 <*asm/page.h*>)中定义的常数`PAGE_SIZE` (给出了任何给定架构上的页面大小。

如果您查看内存地址（虚拟或物理地址），则可以在页面数字中分解为页面和偏移。例如，如果使用4096-Byte页面，例如，12个最不重要的位是偏移量，剩下的较高位表示页码。如果丢弃偏移量并将其余的偏移移向右侧，则结果称为*page frame number*（pfn）。转换位以在页面框架编号和地址之间转换是一个相当普遍的操作。宏`PAGE_SHIFT`告诉必须移动多少位才能进行此转换。

高和低记忆

逻辑和内核虚拟地址之间的区别在配备大量内存的32位系统上突出显示。使用32位，可以解决4 GB的内存。直到最近，在32位系统上的Linux仍将其记忆力少得多，但是由于它设置虚拟地址空间的方式。

内核（在X86体系结构中，在默认配置中）将用户空间和内核之间的4-GB纯正地址空间分开；在这两种情况下都使用了相同的映射。一个典型的拆分将3 GB专用于用户空间，为Kernel空间进行1 GB。*内核的代码和数据结构必须适合该空间，但是内核地址空间的大型消费者是物理内存的虚拟映射。内核无法直接操纵未映射到内核地址空间中的内存。换句话说，内核需要其自己必须直接触摸的任何内存的虚拟地址。因此，多年来，内核可以处理的最大物理记忆量是可以映射到虚拟地址空间内核部分的数量，减去空间

* Many non-x86 architectures are able to efficiently do without the kernel/user-space split described here, so they can work with up to a 4-GB kernel address space on 32-bit systems. The constraints described in this section still apply to such systems when more than 4 GB of memory are installed, however.

needed for the kernel code itself. As a result, x86-based Linux systems could work with a maximum of a little under 1 GB of physical memory.

In response to commercial pressure to support more memory while not breaking 32-bit application and the system's compatibility, the processor manufacturers have added "address extension" features to their products. The result is that, in many cases, even 32-bit processors can address more than 4 GB of physical memory. The limitation on how much memory can be directly mapped with logical addresses remains, however. Only the lowest portion of memory (up to 1 or 2 GB, depending on the hardware and the kernel configuration) has logical addresses;* the rest (high memory) does not. Before accessing a specific high-memory page, the kernel must set up an explicit virtual mapping to make that page available in the kernel's address space. Thus, many kernel data structures must be placed in low memory; high memory tends to be reserved for user-space process pages.

The term "high memory" can be confusing to some, especially since it has other meanings in the PC world. So, to make things clear, we'll define the terms here:

Low memory
: Memory for which logical addresses exist in kernel space. On almost every system you will likely encounter, all memory is low memory.

High memory
: Memory for which logical addresses do not exist, because it is beyond the address range set aside for kernel virtual addresses.

On i386 systems, the boundary between low and high memory is usually set at just under 1 GB, although that boundary can be changed at kernel configuration time. This boundary is not related in any way to the old 640 KB limit found on the original PC, and its placement is not dictated by the hardware. It is, instead, a limit set by the kernel itself as it splits the 32-bit address space between kernel and user space.

We will point out limitations on the use of high memory as we come to them in this chapter.

The Memory Map and Struct Page

Historically, the kernel has used logical addresses to refer to pages of physical memory. The addition of high-memory support, however, has exposed an obvious problem with that approach—logical addresses are not available for high memory. Therefore, kernel functions that deal with memory are increasingly using pointers to `struct page` (defined in *<linux/mm.h>*) instead. This data structure is used to keep track of just about everything the kernel needs to know about physical memory;

* The 2.6 kernel (with an added patch) can support a "4G/4G" mode on x86 hardware, which enables larger kernel and user virtual address spaces at a mild performance cost.

内核代码本身需要。结果，基于X86的Linux系统最多可以使用1 GB的物理内存。

为了响应商业压力，以支持更多的内存，同时不破坏32位应用和系统的兼容性，因此处理器制造商在其产品中添加了“地址扩展”功能。结果是，在许多情况下，即使是32位处理器也可以解决4 GB以上的物理内存。但是，仍然可以直接映射到逻辑地址的多少内存的限制。只有内存的最低部分（最多1或2 GB，具体取决于硬件和内核配置）具有逻辑地址; *剩余（高内存）没有。在访问特定的高内存页面之前，内核必须设置一个明确的虚拟映射，以使该页面在内核的地址空间中可用。因此，许多内核数据结构必须放置在低记忆中。高内存倾向于保留用于用户空间过程页面。

“高内存”一词可能会使某些人感到困惑，尤其是因为它在PC世界中还有其他含义。因此，为了清楚情况，我们将在此处定义术语:

Low memory
: 内核空间中存在逻辑地址的内存。在您可能会遇到的几乎每个系统中，所有内存都是低内存。

High memory
: 不存在逻辑地址的内存，因为它超出了内核虚拟地址的地址范围。

在i386系统上，尽管可以在内核配置时间更改该边界，但低内存和高内存之间的边界通常设置为1 GB。该边界与原始PC上的旧640 KB限制的任何方式无关，并且其位置并不由硬件决定。相反，它是内核本身设定的限制，因为它将内核和用户空间之间的32位地址空间分开。

我们将指出对本章提出的高内存使用的限制。

内存图和结构页面

从历史上看，内核使用逻辑地址来指代物理成员的页面。但是，高内存支持的添加已经暴露了一种明显的概率，而这种方法不可用于高记忆。因此，与 <*linux/mm.h*>)中定义的`struct page` (相反，越来越多地将处理内存的内核函数使用。该数据结构用于跟踪内核所需了解的有关物理内存的所有内容;

* The 2.6 kernel (with an added patch) can support a “4G/4G” mode on x86 hardware, which enables larger kernel and user virtual address spaces at a mild performance cost.

there is one `struct page` for each physical page on the system. Some of the fields of this structure include the following:

`atomic_t count;`
: The number of references there are to this page. When the count drops to `0`, the page is returned to the free list.

`void *virtual;`
: The kernel virtual address of the page, if it is mapped; `NULL`, otherwise. Low-memory pages are always mapped; high-memory pages usually are not. This field does not appear on all architectures; it generally is compiled only where the kernel virtual address of a page cannot be easily calculated. If you want to look at this field, the proper method is to use the *page_address* macro, described below.

`unsigned long flags;`
: A set of bit flags describing the status of the page. These include `PG_locked`, which indicates that the page has been locked in memory, and `PG_reserved`, which prevents the memory management system from working with the page at all.

There is much more information within `struct page`, but it is part of the deeper black magic of memory management and is not of concern to driver writers.

The kernel maintains one or more arrays of `struct page` entries that track all of the physical memory on the system. On some systems, there is a single array called `mem_map`. On some systems, however, the situation is more complicated. Nonuniform memory access (NUMA) systems and those with widely discontiguous physical memory may have more than one memory map array, so code that is meant to be portable should avoid direct access to the array whenever possible. Fortunately, it is usually quite easy to just work with `struct page` pointers without worrying about where they come from.

Some functions and macros are defined for translating between `struct page` pointers and virtual addresses:

`struct page *virt_to_page(void *kaddr);`
: This macro, defined in *<asm/page.h>*, takes a kernel logical address and returns its associated `struct page` pointer. Since it requires a logical address, it does not work with memory from *vmalloc* or high memory.

`struct page *pfn_to_page(int pfn);`
: Returns the `struct page` pointer for the given page frame number. If necessary, it checks a page frame number for validity with *pfn_valid* before passing it to *pfn_to_page*.

`void *page_address(struct page *page);`
: Returns the kernel virtual address of this page, if such an address exists. For high memory, that address exists only if the page has been mapped. This function is

系统上的每个物理页面都有一个struct page。该结构的某些字段包括以下内容：

atomic_t count;
该页面的参考数量。当计数降至0时，该页面将返回到免费列表。

void *virtual;
如果映射的页面的内核虚拟地址；NULL，否则。始终映射低内存页面；高内存页面通常不是。该字段并非出现在所有架构上。通常仅在无法轻松计算页面的内核虚拟地址的情况下进行编译。如果要查看此字段，则适当的方法是使用*page_address*宏，如下所述。

unsigned long flags;
一组描述页面状态的位标志。其中包括PG_locked，该PG_locked表明该页面已锁定在内存中，PG_reserved，它完全阻止了内存管理系统与页面的工作。

struct page中有更多信息，但它是内存管理的深黑魔法的一部分，对驾驶员作家不关心。

内核维护一个或多个struct page条目的数组，该条目跟踪系统上的所有物理内存。在某些系统上，有一个称为mem_map的单个数组。但是，在某些系统上，情况更加复杂。Norrior内存访问（NUMA）系统和具有广泛不连续的物理内存的系统可能具有多个内存映射数组，因此应尽可能避免可移植的代码避免直接访问数组。幸运的是，通常不必担心它们来自何处的struct page指针通常很容易。

定义了一些功能和宏，用于在struct page指针和虚拟地址之间翻译：

struct page *virt_to_page(void *kaddr);
该宏定义在*<asm/page.h>*中，采用一个内核逻辑地址，并返回其关联的struct page指针。由于它需要一个逻辑地址，因此它与*vmalloc*或高内存的内存无效。

struct page *pfn_to_page(int pfn);
返回给定页帧号的struct page指针。如有必要，它在将其传递给*pfn_to_page*之前检查了一个页面框架的有效性。

void *page_address(struct page *page);
如果存在这样的地址，则返回此页面的内核虚拟地址。对于高内存，仅在映射页面时才存在该地址。此功能是

defined in *<linux/mm.h>*. In most situations, you want to use a version of *kmap* rather than *page_address*.

```
#include <linux/highmem.h>
void *kmap(struct page *page);
void kunmap(struct page *page);
```

kmap returns a kernel virtual address for any page in the system. For low-memory pages, it just returns the logical address of the page; for high-memory pages, *kmap* creates a special mapping in a dedicated part of the kernel address space. Mappings created with *kmap* should always be freed with *kunmap*; a limited number of such mappings is available, so it is better not to hold on to them for too long. *kmap* calls maintain a counter, so if two or more functions both call *kmap* on the same page, the right thing happens. Note also that *kmap* can sleep if no mappings are available.

```
#include <linux/highmem.h>
#include <asm/kmap_types.h>
void *kmap_atomic(struct page *page, enum km_type type);
void kunmap_atomic(void *addr, enum km_type type);
```

kmap_atomic is a high-performance form of *kmap*. Each architecture maintains a small list of slots (dedicated page table entries) for atomic kmaps; a caller of *kmap_atomic* must tell the system which of those slots to use in the `type` argument. The only slots that make sense for drivers are `KM_USER0` and `KM_USER1` (for code running directly from a call from user space), and `KM_IRQ0` and `KM_IRQ1` (for interrupt handlers). Note that atomic kmaps must be handled atomically; your code cannot sleep while holding one. Note also that nothing in the kernel keeps two functions from trying to use the same slot and interfering with each other (although there is a unique set of slots for each CPU). In practice, contention for atomic kmap slots seems to not be a problem.

We see some uses of these functions when we get into the example code, later in this chapter and in subsequent chapters.

Page Tables

On any modern system, the processor must have a mechanism for translating virtual addresses into its corresponding physical addresses. This mechanism is called a *page table*; it is essentially a multilevel tree-structured array containing virtual-to-physical mappings and a few associated flags. The Linux kernel maintains a set of page tables even on architectures that do not use such tables directly.

A number of operations commonly performed by device drivers can involve manipulating page tables. Fortunately for the driver author, the 2.6 kernel has eliminated any need to work with page tables directly. As a result, we do not describe them in any detail; curious readers may want to have a look at *Understanding The Linux Kernel* by Daniel P. Bovet and Marco Cesati (O'Reilly) for the full story.

在<*linux/mm.h*>中定义。在大多数情况下，您要使用*kmap*的版本而不是*page_address*。

```
#include <linux/highmem.h>
void *kmap(struct page *page);
void kunmap(struct page *page);
```

kmap 返回系统中任何页面的内核虚拟地址。对于低模拟页面，它只是返回页面的逻辑地址；对于高内存页面，*kmap*在内核地址空间的专用部分中创建一个特殊的映射。用*kmap*创建的映射应始终用*kunmap*释放;有限的此类映射可用，因此最好不要坚持太久。 *kmap*调用维护计数器，因此，如果两个或多个函数在同一页面上均调用*kmap*，则会发生正确的事情。还请注意，如果没有映射，*kmap*可以睡觉。

```
#include <linux/highmem.h>
#include <asm/kmap_types.h>
void *kmap_atomic(struct page *page, enum km_type type);
void kunmap_atomic(void *addr, enum km_type type);
```

kmap_atomic 是*kmap*的高性能形式。每个体系结构都保留了原子kmap的小插槽（专用页表条目）； *kmap_atomic*的呼叫者必须告诉系统在type论证中使用哪个插槽。对于从用户空间)的调用而直接运行的代码，对于驱动程序而言，唯一有意义的插槽是KM_USER0和KM_USER1 (，以及KM_IRQ0和KM_IRQ1 (的中断处理程序)的代码。请注意，原子kmap必须在原子上处理；您的代码在持有一个时无法入睡。还要注意，内核中没有任何东西可以阻止两个函数尝试使用相同的插槽并彼此干扰（尽管每个CPU都有一组唯一的插槽）。实际上，原子KMAP插槽的争论似乎不是问题。

当我们进入示例代码时，在本章和后续章节中，我们会看到这些功能的一些用途。

页表

在任何现代系统上，处理器必须具有将虚拟地址转换为相应的物理地址的机制。该机制称为*page table*;它本质上是一个多级树结构的数组，其中包含虚拟到物理映射和一些相关标志。 Linux内核即使在不直接使用此类表的架构上也保持了一组页面表。

设备驱动程序通常执行的许多操作都涉及操纵页面表。幸运的是，对于驾驶员作者，2.6内核无需直接使用页面表。结果，我们没有详细描述它们。奇怪的读者可能希望查看Daniel P. Bovet和Marco Cesati（O’Reilly）的*Understanding The Linux Ker- nel*的完整故事。

Virtual Memory Areas

The virtual memory area (VMA) is the kernel data structure used to manage distinct regions of a process's address space. A VMA represents a homogeneous region in the virtual memory of a process: a contiguous range of virtual addresses that have the same permission flags and are backed up by the same object (a file, say, or swap space). It corresponds loosely to the concept of a "segment," although it is better described as "a memory object with its own properties." The memory map of a process is made up of (at least) the following areas:

- An area for the program's executable code (often called text)
- Multiple areas for data, including initialized data (that which has an explicitly assigned value at the beginning of execution), uninitialized data (BSS),[*] and the program stack
- One area for each active memory mapping

The memory areas of a process can be seen by looking in */proc/<pid/maps>* (in which *pid*, of course, is replaced by a process ID). */proc/self* is a special case of */proc/pid*, because it always refers to the current process. As an example, here are a couple of memory maps (to which we have added short comments in italics):

```
# cat /proc/1/maps    look at init
08048000-0804e000 r-xp 00000000 03:01 64652      /sbin/init  text
0804e000-0804f000 rw-p 00006000 03:01 64652      /sbin/init  data
0804f000-08053000 rwxp 00000000 00:00 0          zero-mapped BSS
40000000-40015000 r-xp 00000000 03:01 96278      /lib/ld-2.3.2.so  text
40015000-40016000 rw-p 00014000 03:01 96278      /lib/ld-2.3.2.so  data
40016000-40017000 rw-p 00000000 00:00 0          BSS for ld.so
42000000-4212e000 r-xp 00000000 03:01 80290      /lib/tls/libc-2.3.2.so  text
4212e000-42131000 rw-p 0012e000 03:01 80290      /lib/tls/libc-2.3.2.so  data
42131000-42133000 rw-p 00000000 00:00 0          BSS for libc
bffff000-c0000000 rwxp 00000000 00:00 0          Stack segment
ffffe000-fffff000 ---p 00000000 00:00 0          vsyscall page

# rsh wolf cat /proc/self/maps  #### x86-64 (trimmed)
00400000-00405000 r-xp 00000000 03:01 1596291     /bin/cat    text
00504000-00505000 rw-p 00004000 03:01 1596291     /bin/cat    data
00505000-00526000 rwxp 00505000 00:00 0                       bss
3252200000-3252214000 r-xp 00000000 03:01 1237890 /lib64/ld-2.3.3.so
3252300000-3252301000 r--p 00100000 03:01 1237890 /lib64/ld-2.3.3.so
3252301000-3252302000 rw-p 00101000 03:01 1237890 /lib64/ld-2.3.3.so
7fbfffe000-7fc0000000 rw-p 7fbfffe000 00:00 0                 stack
ffffffffff600000-ffffffffffe00000 ---p 00000000 00:00 0       vsyscall
```

The fields in each line are:

```
start-end perm offset major:minor inode image
```

[*] The name *BSS* is a historical relic from an old assembly operator meaning "block started by symbol." The BSS segment of executable files isn't stored on disk, and the kernel maps the zero page to the BSS address range.

虚拟内存区域

虚拟内存区域（VMA）是用于管理过程地址空间不同区域的内核数据结构。 VMA代表一个过程的虚拟内存中的一个均匀区域：具有相同权限标志并由相同对象（文件，例如或交换空间）备份的连续的虚拟地址范围。尽管它更好地描述为“具有其自身属性的内存对象”，但它与“段”的概念松散地对应。程序的内存图由（至少）以下区域组成：

- 该程序可执行代码的区域（通常称为文本）
- 数据的多个领域，包括初始化数据（执行开始时具有明确分配的值的数据），非初始化数据（BSS），*和程序堆栈
- 每个主动内存映射的一个区域

通过查看*/proc/<pid/maps>*（，*pid*当然被过程ID)代替*pid*时，可以看到一个过程的内存区域。 */proc/self*是*/proc/pid*的特殊情况，因为它始终指当前过程。例如，这里有几个内存图（我们在其中添加了斜体简短的评论）：

```
# cat /proc/1/maps    look at init
08048000-0804e000 r-xp 00000000 03:01 64652      /sbin/init  text
0804e000-0804f000 rw-p 00006000 03:01 64652      /sbin/init  data
0804f000-08053000 rwxp 00000000 00:00 0          zero-mapped BSS
40000000-40015000 r-xp 00000000 03:01 96278      /lib/ld-2.3.2.so   text
40015000-40016000 rw-p 00014000 03:01 96278      /lib/ld-2.3.2.so   data
40016000-40017000 rw-p 00000000 00:00 0          BSS for ld.so
42000000-4212e000 r-xp 00000000 03:01 80290      /lib/tls/libc-2.3.2.so  text
4212e000-42131000 rw-p 0012e000 03:01 80290      /lib/tls/libc-2.3.2.so  data
42131000-42133000 rw-p 00000000 00:00 0          BSS for libc
bffff000-c0000000 rwxp 00000000 00:00 0          Stack segment
ffffe000-fffff000 ---p 00000000 00:00 0          vsyscall page

# rsh wolf cat /proc/self/maps  #### x86-64 (trimmed)
00400000-00405000 r-xp 00000000 03:01 1596291     /bin/cat    text
00504000-00505000 rw-p 00004000 03:01 1596291     /bin/cat    data
00505000-00526000 rwxp 00505000 00:00 0                       bss
3252200000-3252214000 r-xp 00000000 03:01 1237890 /lib64/ld-2.3.3.so
3252300000-3252301000 r--p 00100000 03:01 1237890 /lib64/ld-2.3.3.so
3252301000-3252302000 rw-p 00101000 03:01 1237890 /lib64/ld-2.3.3.so
7fbfffe000-7fc0000000 rw-p 7fbfffe000 00:00 0                 stack
ffffffffff600000-ffffffffffe00000 ---p 00000000 00:00 0       vsyscall
```

每行的字段是：

```
start-end perm offset major:minor inode image
```

* The name *BSS* is a historical relic from an old assembly operator meaning “block started by symbol.” The BSS segment of executable files isn’t stored on disk, and the kernel maps the zero page to the BSS address range.

Each field in */proc/*/maps* (except the image name) corresponds to a field in `struct vm_area_struct`:

`start`
`end`
: The beginning and ending virtual addresses for this memory area.

`perm`
: A bit mask with the memory area's read, write, and execute permissions. This field describes what the process is allowed to do with pages belonging to the area. The last character in the field is either `p` for "private" or `s` for "shared."

`offset`
: Where the memory area begins in the file that it is mapped to. An offset of `0` means that the beginning of the memory area corresponds to the beginning of the file.

`major`
`minor`
: The major and minor numbers of the device holding the file that has been mapped. Confusingly, for device mappings, the major and minor numbers refer to the disk partition holding the device special file that was opened by the user, and not the device itself.

`inode`
: The inode number of the mapped file.

`image`
: The name of the file (usually an executable image) that has been mapped.

The vm_area_struct structure

When a user-space process calls *mmap* to map device memory into its address space, the system responds by creating a new VMA to represent that mapping. A driver that supports *mmap* (and, thus, that implements the *mmap* method) needs to help that process by completing the initialization of that VMA. The driver writer should, therefore, have at least a minimal understanding of VMAs in order to support *mmap*.

Let's look at the most important fields in `struct vm_area_struct` (defined in *<linux/mm.h>*). These fields may be used by device drivers in their *mmap* implementation. Note that the kernel maintains lists and trees of VMAs to optimize area lookup, and several fields of `vm_area_struct` are used to maintain this organization. Therefore, VMAs can't be created at will by a driver, or the structures break. The main fields of

/proc//maps* (中的每个字段除外，图像名称)对应于struct vm_area_struct中的一个字段:

start
end
: 此内存区域的开始和结束虚拟地址。

perm
: 内存区域的读，写和执行权限的掩盖。该字段描述了允许该过程对属于该区域的页面进行的操作。该字段中的最后一个字符是“私有”的p或“共享”的s。

offset
: 内存区域从映射到的文件中开始的位置。 0的偏移意味着内存区域的开始对应于文件的开始。

major
minor
: 持有已映射的文件的设备的主要和次要数量。令人困惑的是，对于设备映射，主要数字和次要数字是指用户打开的设备特殊文件的磁盘分区，而不是设备本身。

inode
: 映射文件的inode编号。

image
: 已映射的文件的名称（通常是可执行的映像）。

vm_area_struct结构

当用户空间进程调用*mmap*将设备存储器映射到其地址空间中时，系统会通过创建新的VMA来响应以表示该映射。支持*mmap* (的驱动程序，因此，实现*mmap*方法)的驱动程序需要通过完成该VMA的初始化来帮助该过程。因此，驾驶员作者至少应该对VMA至少了解*mmap*。

让我们看一下<*linux/ mm.h*>)中定义的struct vm_area_struct (中最重要的字段。这些字段可以由设备驱动程序在其*mmap*实现中使用。请注意，内核维护VMA的列表和树以优化区域查找，并且使用了vm_area_struct的几个字段来维护该组织。因此，驾驶员或结构破裂不能随意创建VMA。主要领域

VMAs are as follows (note the similarity between these fields and the */proc* output we just saw):

`unsigned long vm_start;`
`unsigned long vm_end;`
: The virtual address range covered by this VMA. These fields are the first two fields shown in */proc/*/maps*.

`struct file *vm_file;`
: A pointer to the `struct file` structure associated with this area (if any).

`unsigned long vm_pgoff;`
: The offset of the area in the file, in pages. When a file or device is mapped, this is the file position of the first page mapped in this area.

`unsigned long vm_flags;`
: A set of flags describing this area. The flags of the most interest to device driver writers are `VM_IO` and `VM_RESERVED`. `VM_IO` marks a VMA as being a memory-mapped I/O region. Among other things, the `VM_IO` flag prevents the region from being included in process core dumps. `VM_RESERVED` tells the memory management system not to attempt to swap out this VMA; it should be set in most device mappings.

`struct vm_operations_struct *vm_ops;`
: A set of functions that the kernel may invoke to operate on this memory area. Its presence indicates that the memory area is a kernel "object," like the `struct file` we have been using throughout the book.

`void *vm_private_data;`
: A field that may be used by the driver to store its own information.

Like `struct vm_area_struct`, the `vm_operations_struct` is defined in <*linux/mm.h*>; it includes the operations listed below. These operations are the only ones needed to handle the process's memory needs, and they are listed in the order they are declared. Later in this chapter, some of these functions are implemented.

`void (*open)(struct vm_area_struct *vma);`
: The *open* method is called by the kernel to allow the subsystem implementing the VMA to initialize the area. This method is invoked any time a new reference to the VMA is made (when a process forks, for example). The one exception happens when the VMA is first created by *mmap*; in this case, the driver's *mmap* method is called instead.

`void (*close)(struct vm_area_struct *vma);`
: When an area is destroyed, the kernel calls its *close* operation. Note that there's no usage count associated with VMAs; the area is opened and closed exactly once by each process that uses it.

VMA如下（请注意这些字段与我们刚刚看到的*/proc*输出之间的相似性）：

`unsigned long vm_start;`
`unsigned long vm_end;`
: 该VMA涵盖的虚拟地址范围。这些字段是*/proc/*/maps*中显示的前两个字段。

`struct file *vm_file;`
: 指向与此区域关联的`struct file`结构的指针（如果有）。

`unsigned long vm_pgoff;`
: 文件中的区域的偏移，页面。映射文件或设备时，这是该区域中首页映射的文件位置。

`unsigned long vm_flags;`
: 一组描述该区域的标志。设备驱动程序作者最感兴趣的标志是`VM_IO`和`VM_RESERVED`。`VM_IO`将VMA标记为内存映射的I/O区域。除其他外，`VM_IO`标志可防止该区域包含在过程核心转储中。`VM_RESERVED`告诉内存管理系统不要尝试交换此VMA；应该在大多数设备映射中设置。

`struct vm_operations_struct *vm_ops;`
: 内核可以调用以在此内存区域操作的一组功能。它的存在表明内存区域是一个内核“对象”，就像我们在整本书中都使用的`struct file`一样。

`void *vm_private_data;`
: 一个菲驶员可能使用的LD存储自己的Informati 在。

像`struct vm_area_struct`一样，`vm_operations_struct`在<*linux/mm.h*>中定义;它包括以下列出的操作。这些操作是处理过程内存需求的唯一操作，并且按声明的顺序列出了这些操作。在本章的后面，其中一些功能将实现。

`void (*open)(struct vm_area_struct *vma);`
: 内核调用*open*方法允许实现VMA的子系统初始化区域。在任何对VMA进行新的引用时（例如，当过程分叉）的任何新参考时，都会调用此方法。当VMA首先由*mmap*创建时，就会发生一个例外；在这种情况下，驱动程序的*mmap*方法被称为。

`void (*close)(struct vm_area_struct *vma);`
: 当一个区域被摧毁时，内核称其*close*操作。请注意，与VMA相关的用法计数；通过使用它的每个过程，该区域被打开并精确地关闭一次。

`struct page *(*nopage)(struct vm_area_struct *vma, unsigned long address, int *type);`

When a process tries to access a page that belongs to a valid VMA, but that is currently not in memory, the *nopage* method is called (if it is defined) for the related area. The method returns the `struct page` pointer for the physical page after, perhaps, having read it in from secondary storage. If the *nopage* method isn't defined for the area, an empty page is allocated by the kernel.

`int (*populate)(struct vm_area_struct *vm, unsigned long address, unsigned long len, pgprot_t prot, unsigned long pgoff, int nonblock);`

This method allows the kernel to "prefault" pages into memory before they are accessed by user space. There is generally no need for drivers to implement the *populate* method.

The Process Memory Map

The final piece of the memory management puzzle is the process memory map structure, which holds all of the other data structures together. Each process in the system (with the exception of a few kernel-space helper threads) has a `struct mm_struct` (defined in *<linux/sched.h>*) that contains the process's list of virtual memory areas, page tables, and various other bits of memory management housekeeping information, along with a semaphore (`mmap_sem`) and a spinlock (`page_table_lock`). The pointer to this structure is found in the task structure; in the rare cases where a driver needs to access it, the usual way is to use `current->mm`. Note that the memory management structure can be shared between processes; the Linux implementation of threads works in this way, for example.

That concludes our overview of Linux memory management data structures. With that out of the way, we can now proceed to the implementation of the *mmap* system call.

The mmap Device Operation

Memory mapping is one of the most interesting features of modern Unix systems. As far as drivers are concerned, memory mapping can be implemented to provide user programs with direct access to device memory.

A definitive example of *mmap* usage can be seen by looking at a subset of the virtual memory areas for the X Window System server:

```
cat /proc/731/maps
000a0000-000c0000 rwxs 000a0000 03:01 282652      /dev/mem
000f0000-00100000 r-xs 000f0000 03:01 282652      /dev/mem
00400000-005c0000 r-xp 00000000 03:01 1366927     /usr/X11R6/bin/Xorg
006bf000-006f7000 rw-p 001bf000 03:01 1366927     /usr/X11R6/bin/Xorg
2a95828000-2a958a8000 rw-s fcc00000 03:01 282652  /dev/mem
2a958a8000-2a9d8a8000 rw-s e8000000 03:01 282652  /dev/mem
...
```

`struct page *(*nopage)(struct vm_area_struct *vma, unsigned long address, int *type);`

当一个进程试图访问属于有效VMA的页面，但目前不在内存中时，相关区域的*nopage*方法被调用（如果是定义）。该方法返回了物理页面的`struct page`指针，也许是从辅助存储中读取的。如果未针对该区域定义*nopage*方法，则内核分配一个空页面。

`int (*populate)(struct vm_area_struct *vm, unsigned long address, unsigned long len, pgprot_t prot, unsigned long pgoff, int nonblock);`

此方法允许内核在用户空间访问内存之前将它们“预处理”到内存中。通常不需要驱动程序实现*populate*方法。

过程内存图

内存管理难题的最后一部分是过程存储映射结构，它将所有其他数据结构固定在一起。 sys-tem（除几个内核空间辅助线程）中的每个过程都有一个`struct mm_struct` (在<*linux/sched.h*>)中定义的，其中包含该过程的虚拟内存区域，页面表和其他各种内存管理内部保存的内存内部保管员的列表，以及Spinlock（v11}）和一个Spinlock（}）{）{）在任务结构中找到了指向此结构的指针；在极少数需要访问它的情况下，通常的方法是使用`current->mm`。请注意，可以在过程之间共享内存结构；例如，线程的Linux实现以这种方式工作。

这是我们对Linux内存管理数据结构的概述。这样一来，我们现在可以继续实现*mmap*系统调用。

MMAP设备操作

内存映射是现代Unix系统最有趣的功能之一。就驱动程序而言，可以实现内存映射以提供直接访问设备内存的用户程序。

通过查看X Window System Server的虚拟内存区域的子集：

```
cat /proc/731/maps
000a0000-000c0000 rwxs 000a0000 03:01 282652      /dev/mem
000f0000-00100000 r-xs 000f0000 03:01 282652      /dev/mem
00400000-005c0000 r-xp 00000000 03:01 1366927     /usr/X11R6/bin/Xorg
006bf000-006f7000 rw-p 001bf000 03:01 1366927     /usr/X11R6/bin/Xorg
2a95828000-2a958a8000 rw-s fcc00000 03:01 282652  /dev/mem
2a958a8000-2a9d8a8000 rw-s e8000000 03:01 282652  /dev/mem
...
```

The full list of the X server's VMAs is lengthy, but most of the entries are not of interest here. We do see, however, four separate mappings of */dev/mem*, which give some insight into how the X server works with the video card. The first mapping is at a0000, which is the standard location for video RAM in the 640-KB ISA hole. Further down, we see a large mapping at e8000000, an address which is above the highest RAM address on the system. This is a direct mapping of the video memory on the adapter.

These regions can also be seen in */proc/iomem*:

```
000a0000-000bffff : Video RAM area
000c0000-000ccfff : Video ROM
000d1000-000d1fff : Adapter ROM
000f0000-000fffff : System ROM
d7f00000-f7efffff : PCI Bus #01
  e8000000-efffffff : 0000:01:00.0
fc700000-fccfffff : PCI Bus #01
  fcc00000-fcc0ffff : 0000:01:00.0
```

Mapping a device means associating a range of user-space addresses to device memory. Whenever the program reads or writes in the assigned address range, it is actually accessing the device. In the X server example, using *mmap* allows quick and easy access to the video card's memory. For a performance-critical application like this, direct access makes a large difference.

As you might suspect, not every device lends itself to the *mmap* abstraction; it makes no sense, for instance, for serial ports and other stream-oriented devices. Another limitation of *mmap* is that mapping is PAGE_SIZE grained. The kernel can manage virtual addresses only at the level of page tables; therefore, the mapped area must be a multiple of PAGE_SIZE and must live in physical memory starting at an address that is a multiple of PAGE_SIZE. The kernel forces size granularity by making a region slightly bigger if its size isn't a multiple of the page size.

These limits are not a big constraint for drivers, because the program accessing the device is device dependent anyway. Since the program must know about how the device works, the programmer is not unduly bothered by the need to see to details like page alignment. A bigger constraint exists when ISA devices are used on some non-x86 platforms, because their hardware view of ISA may not be contiguous. For example, some Alpha computers see ISA memory as a scattered set of 8-bit, 16-bit, or 32-bit items, with no direct mapping. In such cases, you can't use *mmap* at all. The inability to perform direct mapping of ISA addresses to Alpha addresses is due to the incompatible data transfer specifications of the two systems. Whereas early Alpha processors could issue only 32-bit and 64-bit memory accesses, ISA can do only 8-bit and 16-bit transfers, and there's no way to transparently map one protocol onto the other.

There are sound advantages to using *mmap* when it's feasible to do so. For instance, we have already looked at the X server, which transfers a lot of data to and from

X服务器VMA的完整列表很长，但是大多数条目在这里不涉及。但是，我们确实看到了*/dev/mem*的四个单独的映射，这些映射可以深入了解X服务器如何与视频卡一起使用。第一个映射位于a0000，这是640-KB ISA孔中视频RAM的标准位置。在此外，我们看到e8000000的大型映射，该地址高于系统上最高的RAM地址。这是适配器上视频内存的直接映射。

这些区域也可以在*/proc/iomem*中看到：

```
000a0000-000bffff : Video RAM area
000c0000-000ccfff : Video ROM
000d1000-000d1fff : Adapter ROM
000f0000-000fffff : System ROM
d7f00000-f7efffff : PCI Bus #01
  e8000000-efffffff : 0000:01:00.0
fc700000-fccfffff : PCI Bus #01
  fcc00000-fcc0ffff : 0000:01:00.0
```

映射设备意味着将一系列用户空间地址与设备mem-Ory相关联。每当程序在分配的地址范围内读取或写入时，它都会访问设备。在X服务器示例中，使用*mmap*可以快速轻松地访问视频卡的内存。对于这样的关键绩效应用程序，直接访问会带来很大的不同。

您可能会怀疑，并非每个设备都将自己借给*mmap*抽象；例如，对于串行端口和其他面向流的设备，这是没有意义的。 *mmap*的另一个限制是映射为PAGE_SIZE粒度。内核只能在页面表的级别上管理Virtual地址；因此，映射的区域必须是PAGE_SIZE的倍数，并且必须生活在物理内存中，从PAGE_SIZE的一个倍数开始。如果内核大小粒度使区域的大小不是页面尺寸的倍数，则力量大小粒度。

这些限制对驱动程序并不是一个很大的限制，因为无论如何，访问设备的程序都是依赖设备的。由于该程序必须了解设备的工作原理，因此编程器并不是要对诸如页面对齐之类的详细信息的需求过分困扰。当在某些非X86平台上使用ISA设备时，存在更大的约束，因为它们的硬件视图可能不连续。例如，某些Alpha计算机将ISA存储器视为一组散射的8位，16位或32位项目，而没有直接映射。在这种情况下，您根本不能使用*mmap*。无法将ISA地址直接映射到Alpha地址的直接映射是由于两个系统的不兼容数据传输规范所致。早期的Alpha处理器只能发布32位和64位内存访问，而ISA只能进行8位和16位的转移，并且无法将一个原始的一个原型绘制到另一个原始范围。

当它可行时，使用*mmap*有合理的优势。例如，我们已经查看了X服务器，该服务器将大量数据传输到往返

video memory; mapping the graphic display to user space dramatically improves the throughput, as opposed to an *lseek*/*write* implementation. Another typical example is a program controlling a PCI device. Most PCI peripherals map their control registers to a memory address, and a high-performance application might prefer to have direct access to the registers instead of repeatedly having to call *ioctl* to get its work done.

The *mmap* method is part of the `file_operations` structure and is invoked when the *mmap* system call is issued. With *mmap*, the kernel performs a good deal of work before the actual method is invoked, and, therefore, the prototype of the method is quite different from that of the system call. This is unlike calls such as *ioctl* and *poll*, where the kernel does not do much before calling the method.

The system call is declared as follows (as described in the *mmap(2)* manual page):

```
mmap (caddr_t addr, size_t len, int prot, int flags, int fd, off_t offset)
```

On the other hand, the file operation is declared as:

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

The `filp` argument in the method is the same as that introduced in Chapter 3, while `vma` contains the information about the virtual address range that is used to access the device. Therefore, much of the work has been done by the kernel; to implement *mmap*, the driver only has to build suitable page tables for the address range and, if necessary, replace `vma->vm_ops` with a new set of operations.

There are two ways of building the page tables: doing it all at once with a function called `remap_pfn_range` or doing it a page at a time via the *nopage* VMA method. Each method has its advantages and limitations. We start with the "all at once" approach, which is simpler. From there, we add the complications needed for a real-world implementation.

Using remap_pfn_range

The job of building new page tables to map a range of physical addresses is handled by *remap_pfn_range* and *io_remap_page_range*, which have the following prototypes:

```
int remap_pfn_range(struct vm_area_struct *vma,
                     unsigned long virt_addr, unsigned long pfn,
                     unsigned long size, pgprot_t prot);
int io_remap_page_range(struct vm_area_struct *vma,
                        unsigned long virt_addr, unsigned long phys_addr,
                        unsigned long size, pgprot_t prot);
```

视频记忆；将图形显示映射到用户空间可以显着改善吞吐量，而不是*lseek*/*write*实现。另一个典型的示例是控制PCI设备的程序。大多数PCI外围设备将其控件重新分配到存储地址，并且高性能应用程序可能更喜欢直接访问寄存器，而不是反复致电*ioctl*才能完成其工作。

*mmap*方法是`file_operations`结构的一部分，并在发出*mmap*系统调用时被调用。使用*mmap*，内核在调用实际方法之前执行了很多工作，因此，该方法的原型与系统调用的原型完全不同。这与*ioctl*和*poll*之类的调用不同，在调用该方法之前，内核没有做太多事情。

系统调用如下所示（如*mmap(2)*手动页面中所述）：

```
mmap (caddr_t addr, size_t len, int prot, int flags, int fd, off_t offset)
```

另一方面，文件操作被声明为：

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

该方法中的`filp`参数与第3章中介绍的参数相同，而`vma`包含有关用于访问设备的虚拟地址范围的信息。因此，大部分工作都是由内核完成的。要实现*mmap*，驱动程序只需要为地址范围构建合适的页面表，并在必要时用新的操作替换`vma->vm_ops`。

构建页面表有两种方法：一次使用称为`remap_pfn_range`的函数一次或通过*nopage* VMA方法执行页面。每种方法都有其优点和局限性。我们从“一次”方法开始，这更简单。从那里开始，我们添加了实施现实世界所需的复杂性。

使用remap_pfn_range

构建新页面表以绘制一系列物理地址的工作由*remap_pfn_range*和*io_remap_page_range*处理，它们具有以下原型：

```
int remap_pfn_range(struct vm_area_struct *vma,
                     unsigned long virt_addr, unsigned long pfn,
                     unsigned long size, pgprot_t prot);
int io_remap_page_range(struct vm_area_struct *vma,
                        unsigned long virt_addr, unsigned long phys_addr,
                        unsigned long size, pgprot_t prot);
```

The value returned by the function is the usual `0` or a negative error code. Let's look at the exact meaning of the function's arguments:

`vma`
: The virtual memory area into which the page range is being mapped.

`virt_addr`
: The user virtual address where remapping should begin. The function builds page tables for the virtual address range between `virt_addr` and `virt_addr+size`.

`pfn`
: The page frame number corresponding to the physical address to which the virtual address should be mapped. The page frame number is simply the physical address right-shifted by `PAGE_SHIFT` bits. For most uses, the `vm_pgoff` field of the VMA structure contains exactly the value you need. The function affects physical addresses from `(pfn<<PAGE_SHIFT)` to `(pfn<<PAGE_SHIFT)+size`.

`size`
: The dimension, in bytes, of the area being remapped.

`prot`
: The "protection" requested for the new VMA. The driver can (and should) use the value found in `vma->vm_page_prot`.

The arguments to *remap_pfn_range* are fairly straightforward, and most of them are already provided to you in the VMA when your *mmap* method is called. You may be wondering why there are two functions, however. The first (*remap_pfn_range*) is intended for situations where `pfn` refers to actual system RAM, while *io_remap_page_range* should be used when `phys_addr` points to I/O memory. In practice, the two functions are identical on every architecture except the SPARC, and you see *remap_pfn_range* used in most situations. In the interest of writing portable drivers, however, you should use the variant of *remap_pfn_range* that is suited to your particular situation.

One other complication has to do with caching: usually, references to device memory should not be cached by the processor. Often the system BIOS sets things up properly, but it is also possible to disable caching of specific VMAs via the protection field. Unfortunately, disabling caching at this level is highly processor dependent. The curious reader may wish to look at the *pgprot_noncached* function from *drivers/char/mem.c* to see what's involved. We won't discuss the topic further here.

A Simple Implementation

If your driver needs to do a simple, linear mapping of device memory into a user address space, *remap_pfn_range* is almost all you really need to do the job. The following code is

该函数返回的值是通常的0或负错误代码。让我们看一下该函数论点的确切含义
:

vma
: ThE页面范围为M的虚拟内存区域 应用。

virt_addr
: 用户虚拟地址应开始重新映射。该函数为virt_addr和virt_addr+size之间的虚拟地址构建页面表范围。

pfn
: 页面框架编号对应于应映射的物理地址的物理地址。页面框架号仅是PAGE_SHIFT位右移的物理地址。对于大多数用途，VMA结构的vm_pgoff字段完全包含您需要的值。该功能会影响从(pfn<<PAGE_SHIFT)到(pfn<<PAGE_SHIFT)+size的物理地址。

size
: 该区域被重新映射的尺寸，字节。

prot
: “保护”请求新的VMA。驱动程序可以（并且应该）使用vma->vm_page_prot中的值。

*remap_pfn_range*的参数非常简单，当调用*mmap*方法时，其中大多数已经在VMA中提供给您。但是，您可能想知道为什么有两个功能。第一个（*remap_pfn_range*）用于pfn是指实际系统RAM，而*io_remap_ page_range*指向I/O内存时，应使用*io_remap_ page_range*。实际上，这两个函数在除SPARC以外的每个体系结构上都是相同的，并且您会看到在大多数情况下使用的*remap_pfn_range*。但是，为了撰写便携式驱动程序，您应该使用适合您特定情况的*remap_pfn_range*的变体。

另一个并发症与缓存有关：通常，处理器不应缓存对设备内存的引用。通常，BIOS系统会正确设置物体，但也可以通过Protecection领域禁用特定VMA的缓存。不幸的是，在此级别上禁用缓存是高度的处理器。好奇的读者可能希望从*drivers/char/mem.c*中查看*pgprot_noncached*函数，以查看所涉及的内容。我们不会在这里进一步讨论这个话题。

一个简单的实现

如果您的驱动程序需要对设备内存进行简单的线性映射到用户地址空间中，那么*remap_pfn_range*几乎是您真正需要完成的工作。以下代码是

derived from *drivers/char/mem.c* and shows how this task is performed in a typical module called *simple* (Simple Implementation Mapping Pages with Little Enthusiasm):

```
static int simple_remap_mmap(struct file *filp, struct vm_area_struct *vma)
{
    if (remap_pfn_range(vma, vma->vm_start, vm->vm_pgoff,
                vma->vm_end - vma->vm_start,
                vma->vm_page_prot))
        return -EAGAIN;

    vma->vm_ops = &simple_remap_vm_ops;
    simple_vma_open(vma);
    return 0;
}
```

As you can see, remapping memory just a matter of calling *remap_pfn_range* to create the necessary page tables.

Adding VMA Operations

As we have seen, the `vm_area_struct` structure contains a set of operations that may be applied to the VMA. Now we look at providing those operations in a simple way. In particular, we provide *open* and *close* operations for our VMA. These operations are called whenever a process opens or closes the VMA; in particular, the *open* method is invoked anytime a process forks and creates a new reference to the VMA. The *open* and *close* VMA methods are called in addition to the processing performed by the kernel, so they need not reimplement any of the work done there. They exist as a way for drivers to do any additional processing that they may require.

As it turns out, a simple driver such as *simple* need not do any extra processing in particular. So we have created *open* and *close* methods, which print a message to the system log informing the world that they have been called. Not particularly useful, but it does allow us to show how these methods can be provided, and see when they are invoked.

To this end, we override the default `vma->vm_ops` with operations that call *printk*:

```
void simple_vma_open(struct vm_area_struct *vma)
{
    printk(KERN_NOTICE "Simple VMA open, virt %lx, phys %lx\n",
            vma->vm_start, vma->vm_pgoff << PAGE_SHIFT);
}

void simple_vma_close(struct vm_area_struct *vma)
{
    printk(KERN_NOTICE "Simple VMA close.\n");
}

static struct vm_operations_struct simple_remap_vm_ops = {
    .open =  simple_vma_open,
    .close = simple_vma_close,
};
```

源自*drivers/char/mem.c*，并在典型的模块中显示该任务是如何执行的，称为*simple*（简单的实现映射页面几乎没有热情）：

```
static int simple_remap_mmap(struct file *filp, struct vm_area_struct *vma)
{
    if (remap_pfn_range(vma, vma->vm_start, vm->vm_pgoff,
                vma->vm_end - vma->vm_start,
                vma->vm_page_prot))
        return -EAGAIN;

    vma->vm_ops = &simple_remap_vm_ops;
    simple_vma_open(vma);
    return 0;
}
```

如您所见，仅重新刷新内存的问题是调用*remap_pfn_range*来浏览必要的页面表。

添加VMA操作

如我们所见，vm_area_struct结构包含一组可能应用于VMA的操作。现在，我们考虑以一种简单的方式提供这些操作。特别是，我们为我们的VMA提供*open*和*close*操作。每当过程打开或关闭VMA时，这些操作都会称为这些操作；特别是，随时调用*open*方法，并创建对VMA的新引用。除了内核执行的处理外，还调用了*open*和*close* VMA方法，因此它们无需重新进来。它们的存在是驾驶员进行可能需要的任何其他处理的一种方式。

事实证明，一个简单的驱动程序，例如*simple*，不必特别执行任何额外的处理。因此，我们创建了*open*和*close*方法，该方法向系统日志打印一条消息，告知世界已被调用。并不是特别有用，但是它确实使我们能够展示如何提供这些方法，并查看它们何时被调用。

为此，我们用调用*printk*的操作覆盖默认vma->vm_ops：

```
void simple_vma_open(struct vm_area_struct *vma)
{
    printk(KERN_NOTICE "Simple VMA open, virt %lx, phys %lx\n",
            vma->vm_start, vma->vm_pgoff << PAGE_SHIFT);
}

void simple_vma_close(struct vm_area_struct *vma)
{
    printk(KERN_NOTICE "Simple VMA close.\n");
}

static struct vm_operations_struct simple_remap_vm_ops = {
    .open =  simple_vma_open,
    .close = simple_vma_close,
};
```

To make these operations active for a specific mapping, it is necessary to store a pointer to `simple_remap_vm_ops` in the `vm_ops` field of the relevant VMA. This is usually done in the *mmap* method. If you turn back to the *simple_remap_mmap* example, you see these lines of code:

```
vma->vm_ops = &simple_remap_vm_ops;
simple_vma_open(vma);
```

Note the explicit call to *simple_vma_open*. Since the *open* method is not invoked on the initial *mmap*, we must call it explicitly if we want it to run.

Mapping Memory with nopage

Although *remap_pfn_range* works well for many, if not most, driver *mmap* implementations, sometimes it is necessary to be a little more flexible. In such situations, an implementation using the *nopage* VMA method may be called for.

One situation in which the *nopage* approach is useful can be brought about by the *mremap* system call, which is used by applications to change the bounding addresses of a mapped region. As it happens, the kernel does not notify drivers directly when a mapped VMA is changed by *mremap*. If the VMA is reduced in size, the kernel can quietly flush out the unwanted pages without telling the driver. If, instead, the VMA is expanded, the driver eventually finds out by way of calls to *nopage* when mappings must be set up for the new pages, so there is no need to perform a separate notification. The *nopage* method, therefore, must be implemented if you want to support the *mremap* system call. Here, we show a simple implementation of *nopage* for the *simple* device.

The *nopage* method, remember, has the following prototype:

```
struct page *(*nopage)(struct vm_area_struct *vma,
                       unsigned long address, int *type);
```

When a user process attempts to access a page in a VMA that is not present in memory, the associated *nopage* function is called. The `address` parameter contains the virtual address that caused the fault, rounded down to the beginning of the page. The *nopage* function must locate and return the `struct page` pointer that refers to the page the user wanted. This function must also take care to increment the usage count for the page it returns by calling the *get_page* macro:

```
get_page(struct page *pageptr);
```

This step is necessary to keep the reference counts correct on the mapped pages. The kernel maintains this count for every page; when the count goes to `0`, the kernel knows that the page may be placed on the free list. When a VMA is unmapped, the kernel decrements the usage count for every page in the area. If your driver does not increment the count when adding a page to the area, the usage count becomes `0` prematurely, and the integrity of the system is compromised.

为了使这些操作为特定的映射活动，有必要在相关VMA的vm_ops字段中存储一个指向simple_remap_vm_ops的指针。这是在*mmap*方法中完成的。如果回到*simple_remap_mmap*审查中，您会看到以下代码行：

```
vma->vm_ops = &simple_remap_vm_ops;
simple_vma_open(vma);
```

请注意*simple_vma_open*的明确调用。由于*open*方法未在初始*mmap*上调用，因此，如果希望运行，我们必须明确称其为单位。

用nopage映射内存

尽管*remap_pfn_range*对许多（如果不是大多数）*mmap*的效果很好，有时有时必须更加灵活。在这种情况下，可以要求使用*nopage* VMA方法的实现。

*nopage*方法有用的一种情况可以由*mremap*系统调用带来，应用程序用于更改映射区域的边界地址。碰巧的是，当通过*mremap*更改映射的VMA时，内核不会直接通知驱动程序。如果VMA的尺寸减小，则内核可以在不告诉驾驶员的情况下悄悄地冲出不需要的页面。相反，如果扩展了VMA，则当必须为新页面设置映射时，驱动程序最终通过呼叫*nopage*来查找，因此无需执行单独的通知。因此，如果要支持*mremap*系统调用，则必须实现*nopage*方法。在这里，我们显示了*simple*设备的*nopage*的简单实现。

请记住，*nopage*方法具有以下原型：

```
struct page *(*nopage)(struct vm_area_struct *vma,
                       unsigned long address, int *type);
```

当用户进程尝试访问Mem-Ory中不存在的VMA中的页面时，称为关联的*nopage*函数。 address参数包含导致故障的纯正地址，该地址舍入到页面的开头。 *nopage*函数必须定位并返回struct page指针，该指针指的是用户想要的页面。此功能还必须注意通过调用*get_page*宏：返回的页面的使用计数：

```
get_page(struct page *pageptr);
```

此步骤是必须在映射页面上保持参考计数正确的必要条件。内核维护每个页面的计数；当计数到达0时，内核知道该页面可以放在免费列表上。当VMA未上限时，内核会减少该区域中每个页面的使用计数。如果您的驱动程序在将页面添加到该区域时不会增加计数，则使用使用计数将变为0，并且系统的完整性被损害。

The *nopage* method should also store the type of fault in the location pointed to by the type argument—but only if that argument is not NULL. In device drivers, the proper value for type will invariably be VM_FAULT_MINOR.

If you are using *nopage*, there is usually very little work to be done when *mmap* is called; our version looks like this:

```
static int simple_nopage_mmap(struct file *filp, struct vm_area_struct *vma)
{
    unsigned long offset = vma->vm_pgoff << PAGE_SHIFT;

    if (offset >= __pa(high_memory) || (filp->f_flags & O_SYNC))
        vma->vm_flags |= VM_IO;
    vma->vm_flags |= VM_RESERVED;

    vma->vm_ops = &simple_nopage_vm_ops;
    simple_vma_open(vma);
    return 0;
}
```

The main thing *mmap* has to do is to replace the default (NULL) vm_ops pointer with our own operations. The *nopage* method then takes care of "remapping" one page at a time and returning the address of its struct page structure. Because we are just implementing a window onto physical memory here, the remapping step is simple: we only need to locate and return a pointer to the struct page for the desired address. Our *nopage* method looks like the following:

```
struct page *simple_vma_nopage(struct vm_area_struct *vma,
                unsigned long address, int *type)
{
    struct page *pageptr;
    unsigned long offset = vma->vm_pgoff << PAGE_SHIFT;
    unsigned long physaddr = address - vma->vm_start + offset;
    unsigned long pageframe = physaddr >> PAGE_SHIFT;

    if (!pfn_valid(pageframe))
        return NOPAGE_SIGBUS;
    pageptr = pfn_to_page(pageframe);
    get_page(pageptr);
    if (type)
        *type = VM_FAULT_MINOR;
    return pageptr;
}
```

Since, once again, we are simply mapping main memory here, the *nopage* function need only find the correct struct page for the faulting address and increment its reference count. Therefore, the required sequence of events is to calculate the desired physical address, and turn it into a page frame number by right-shifting it PAGE_SHIFT bits. Since user space can give us any address it likes, we must ensure that we have a valid page frame; the *pfn_valid* function does that for us. If the address is out of range, we return NOPAGE_SIGBUS, which causes a bus signal to be delivered to the calling process.

*nopage*方法还应将故障类型存储在type参数指向的位置中，但只有当该参数不是NULL时，只有。在设备驱动程序中，type的正确值总是VM_FAULT_MINOR。

如果您使用的是*nopage*，则通常几乎没有工作*mmap*。我们的版本看起来像这样：

```
static int simple_nopage_mmap(struct file *filp, struct vm_area_struct *vma)
{
    unsigned long offset = vma->vm_pgoff << PAGE_SHIFT;

    if (offset >= __pa(high_memory) || (filp->f_flags & O_SYNC))
        vma->vm_flags |= VM_IO;
    vma->vm_flags |= VM_RESERVED;

    vma->vm_ops = &simple_nopage_vm_ops;
    simple_vma_open(vma);
    return 0;
}
```

*mmap*的主要内容是用我们自己的操作替换默认值（NULL）vm_ops指针。然后，*nopage*方法一次处理一个页面，并返回其struct page结构的地址。因为我们只是在此处实现一个窗口，所以重新映射步骤很简单：我们只需要找到并返回指针到所需地址的struct page即可。我们的*nopage*方法看起来如下：

```
struct page *simple_vma_nopage(struct vm_area_struct *vma,
                unsigned long address, int *type)
{
    struct page *pageptr;
    unsigned long offset = vma->vm_pgoff << PAGE_SHIFT;
    unsigned long physaddr = address - vma->vm_start + offset;
    unsigned long pageframe = physaddr >> PAGE_SHIFT;

    if (!pfn_valid(pageframe))
        return NOPAGE_SIGBUS;
    pageptr = pfn_to_page(pageframe);
    get_page(pageptr);
    if (type)
        *type = VM_FAULT_MINOR;
    return pageptr;
}
```

由于我们再次在此处简单地映射主内存，因此*nopage*函数只需要找到正确的struct page以便为故障地址找到正确的struct page并增加其参考数量。因此，所需的事件序列是计算所需的物理地址，并通过将其右移动PAGE_SHIFT位将其转换为页面框架。由于用户空间可以为我们提供所喜欢的任何地址，因此我们必须确保我们有一个有效的页面框架； *pfn_valid*函数为我们做到了。如果地址不超出范围，我们返回NOPAGE_SIGBUS，这会导致总线信号传递到呼叫过程。

Otherwise, *pfn_to_page* gets the necessary `struct page` pointer; we can increment its reference count (with a call to *get_page*) and return it.

The *nopage* method normally returns a pointer to a `struct page`. If, for some reason, a normal page cannot be returned (e.g., the requested address is beyond the device's memory region), `NOPAGE_SIGBUS` can be returned to signal the error; that is what the *simple* code above does. *nopage* can also return `NOPAGE_OOM` to indicate failures caused by resource limitations.

Note that this implementation works for ISA memory regions but not for those on the PCI bus. PCI memory is mapped above the highest system memory, and there are no entries in the system memory map for those addresses. Because there is no `struct page` to return a pointer to, *nopage* cannot be used in these situations; you must use *remap_pfn_range* instead.

If the *nopage* method is left `NULL`, kernel code that handles page faults maps the zero page to the faulting virtual address. The *zero page* is a copy-on-write page that reads as `0` and that is used, for example, to map the BSS segment. Any process referencing the zero page sees exactly that: a page filled with zeroes. If the process writes to the page, it ends up modifying a private copy. Therefore, if a process extends a mapped region by calling *mremap*, and the driver hasn't implemented *nopage*, the process ends up with zero-filled memory instead of a segmentation fault.

Remapping Specific I/O Regions

All the examples we've seen so far are reimplementations of */dev/mem*; they remap physical addresses into user space. The typical driver, however, wants to map only the small address range that applies to its peripheral device, not all memory. In order to map to user space only a subset of the whole memory range, the driver needs only to play with the offsets. The following does the trick for a driver mapping a region of `simple_region_size` bytes, beginning at physical address `simple_region_start` (which should be page-aligned):

```
unsigned long off = vma->vm_pgoff << PAGE_SHIFT;
unsigned long physical = simple_region_start + off;
unsigned long vsize = vma->vm_end - vma->vm_start;
unsigned long psize = simple_region_size - off;

if (vsize > psize)
    return -EINVAL; /*  spans too high */
remap_pfn_range(vma, vma_>vm_start, physical, vsize, vma->vm_page_prot);
```

In addition to calculating the offsets, this code introduces a check that reports an error when the program tries to map more memory than is available in the I/O region of the target device. In this code, `psize` is the physical I/O size that is left after the offset has been specified, and `vsize` is the requested size of virtual memory; the function refuses to map addresses that extend beyond the allowed memory range.

否则，*pfn_to_page*获得必要的struct page指针;我们可以将其参考计数（用调用*get_page*）汇总并返回。

*nopage*方法通常返回指针到struct page。如果由于某种原因无法返回普通页面（例如，请求的地址超出了设备的内存区域），则可以返回NOPAGE_SIGBUS以发出错误；这就是上面的*simple*代码所做的。 *nopage*还可以返回NOPAGE_OOM，以指示由资源限制引起的故障。

请注意，此实现适用于ISA内存区域，但对PCI总线上的实现区域不起作用。 PCI内存映射到最高的系统内存上方，并且这些地址的系统内存映射中没有条目。因为在这些情况下不能使用struct page page返回指针。您必须改用*remap_pfn_range*。

如果将*nopage*方法留在NULL中，则处理页面故障的内核代码将零页面映射到故障虚拟地址。 *zero page*是一个抄写页面，读为0，例如用于映射BSS段。引用零页面的任何过程都可以准确地看到：一个填充零的页面。如果该过程写入页面，则最终会修改私有副本。因此，如果一个过程通过调用*mremap*来扩展映射的区域，并且驱动程序尚未实现*nopage*，则该过程最终以零填充内存而不是分段故障。

重建特定的I/O区域

到目前为止，我们看到的所有示例都是*/dev/mem*的重新实现；它们将物理地址重新映射到用户空间中。但是，典型的驱动程序只想映射适用于其外围设备的小地址范围，而不是全部内存。为了将整个内存范围的子集映射到用户空间，驱动程序只需要使用偏移量即可。以下是驱动程序映射simple_region_size字节的区域的功能，从物理地址simple_region_start (开始，该区域应以页面为单位simple_region_start ( ):

```
unsigned long off = vma->vm_pgoff << PAGE_SHIFT;
unsigned long physical = simple_region_start + off;
unsigned long vsize = vma->vm_end - vma->vm_start;
unsigned long psize = simple_region_size - off;

if (vsize > psize)
    return -EINVAL; /*  spans too high */
remap_pfn_range(vma, vma_>vm_start, physical, vsize, vma->vm_page_prot);
```

除了计算偏移外，此代码还引入了一项检查，该检查报告了程序试图映射目标比目标设备的I/O更大的内存时的错误。在此代码中，psize是指定偏置之后留下的物理I/O大小，vsize是虚拟内存的请求大小；该功能拒绝映射超出允许内存范围的地址。

Note that the user process can always use *mremap* to extend its mapping, possibly past the end of the physical device area. If your driver fails to define a *nopage* method, it is never notified of this extension, and the additional area maps to the zero page. As a driver writer, you may well want to prevent this sort of behavior; mapping the zero page onto the end of your region is not an explicitly bad thing to do, but it is highly unlikely that the programmer wanted that to happen.

The simplest way to prevent extension of the mapping is to implement a simple *nopage* method that always causes a bus signal to be sent to the faulting process. Such a method would look like this:

```
struct page *simple_nopage(struct vm_area_struct *vma,
                           unsigned long address, int *type);
{ return NOPAGE_SIGBUS; /* send a SIGBUS */}
```

As we have seen, the *nopage* method is called only when the process dereferences an address that is within a known VMA but for which there is currently no valid page table entry. If we have used *remap_pfn_range* to map the entire device region, the *nopage* method shown here is called only for references outside of that region. Thus, it can safely return `NOPAGE_SIGBUS` to signal an error. Of course, a more thorough implementation of *nopage* could check to see whether the faulting address is within the device area, and perform the remapping if that is the case. Once again, however, *nopage* does not work with PCI memory areas, so extension of PCI mappings is not possible.

Remapping RAM

An interesting limitation of *remap_pfn_range* is that it gives access only to reserved pages and physical addresses above the top of physical memory. In Linux, a page of physical addresses is marked as “reserved” in the memory map to indicate that it is not available for memory management. On the PC, for example, the range between 640 KB and 1 MB is marked as reserved, as are the pages that host the kernel code itself. Reserved pages are locked in memory and are the only ones that can be safely mapped to user space; this limitation is a basic requirement for system stability.

Therefore, *remap_pfn_range* won’t allow you to remap conventional addresses, which include the ones you obtain by calling *get_free_page*. Instead, it maps in the zero page. Everything appears to work, with the exception that the process sees private, zero-filled pages rather than the remapped RAM that it was hoping for. Nonetheless, the function does everything that most hardware drivers need it to do, because it can remap high PCI buffers and ISA memory.

The limitations of *remap_pfn_range* can be seen by running *mapper*, one of the sample programs in *misc-progs* in the files provided on O’Reilly’s FTP site. *mapper* is a simple tool that can be used to quickly test the *mmap* system call; it maps read-only parts of a file specified by command-line options and dumps the mapped region to standard output. The following session, for instance, shows that */dev/mem* doesn’t

请注意，用户进程始终可以使用*mremap*扩展其映射，这可能超过物理设备区域的末端。如果您的驱动程序未能定义*nopage*方法，则永远不会将其通知此扩展名，以及零页面的附加区域地图。作为驾驶员作家，您很可能想防止这种行为；将零页面映射到您区域的尽头是明显的坏事，但是程序员不太可能希望发生这种情况。

防止映射扩展的最简单方法是实现一种简单的*nopage*方法，该方法始终导致总线信号发送到故障过程。这样的方法看起来像：

```
struct page *simple_nopage(struct vm_area_struct *vma,
                           unsigned long address, int *type);
{ return NOPAGE_SIGBUS; /* send a SIGBUS */}
```

如我们所见，仅当进程删除一个已知VMA的地址但目前没有有效的Page Table条目时，*nopage*方法才会调用。如果我们使用*remap_pfn_range*映射整个设备区域，则此处显示的*nopage*方法仅用于该区域之外的引用。因此，它可以安全地返回`NOPAGE_SIGBUS`以发出错误。当然，*nopage*的更彻底的实现可以检查是否在设备区域内的故障地址，并执行重新映射。但是，*nopage*再次不适用于PCI内存区域，因此不可能扩展PCI映射。

重建RAM

*remap_pfn_range*的一个有趣限制是，它仅可访问物理内存顶部上方的保留页面和物理地址。在Linux中，物理地址的一页在内存图中标记为“保留”，以表明它无法用于内存管理。例如，在PC上，在640 Kb和1 MB之间的范围标记为保留，以及主持内核代码本身的页面也是如此。保留页面被锁定在内存中，并且是唯一可以安全地映射到用户空间的页面；此限制是系统稳定性的基本要求。

因此，*remap_pfn_range*将不允许您重建常规地址，其中包括通过调用*get_free_page*获得的地址。相反，它在零页面中映射。一切似乎都起作用了，除了该过程看到的，零填充的页面，而不是它所希望的重新装饰的RAM。尽管如此，该函数可以执行大多数硬件驱动程序需要做的所有操作，因为它可以重塑较高的PCI缓冲区和ISA内存。

*remap_pfn_range*的局限性可以通过运行*mapper*（V16}（V16}）的*mapper*（v17}中的Sample程序之一）中的局限性，在O’Reilly的FTP网站上提供的文件中。*mapper*是一个简单的工具，可用于快速测试*mmap*系统调用；它映射由命令行选项指定的文件的仅读取部分，并将映射的区域转换为标准输出。例如，以下会话表明*/dev/mem*没有

map the physical page located at address 64 KB—instead, we see a page full of zeros (the host computer in this example is a PC, but the result would be the same on other platforms):

```
morgana.root# ./mapper /dev/mem 0x10000 0x1000 | od -Ax -t x1
mapped "/dev/mem" from 65536 to 69632
000000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
*
001000
```

The inability of *remap_pfn_range* to deal with RAM suggests that memory-based devices like *scull* can't easily implement *mmap*, because its device memory is conventional RAM, not I/O memory. Fortunately, a relatively easy workaround is available to any driver that needs to map RAM into user space; it uses the *nopage* method that we have seen earlier.

Remapping RAM with the nopage method

The way to map real RAM to user space is to use `vm_ops->nopage` to deal with page faults one at a time. A sample implementation is part of the *scullp* module, introduced in Chapter 8.

scullp is a page-oriented char device. Because it is page oriented, it can implement *mmap* on its memory. The code implementing memory mapping uses some of the concepts introduced in the section "Memory Management in Linux."

Before examining the code, let's look at the design choices that affect the *mmap* implementation in *scullp*:

- *scullp* doesn't release device memory as long as the device is mapped. This is a matter of policy rather than a requirement, and it is different from the behavior of *scull* and similar devices, which are truncated to a length of `0` when opened for writing. Refusing to free a mapped *scullp* device allows a process to overwrite regions actively mapped by another process, so you can test and see how processes and device memory interact. To avoid releasing a mapped device, the driver must keep a count of active mappings; the `vmas` field in the device structure is used for this purpose.
- Memory mapping is performed only when the *scullp* `order` parameter (set at module load time) is `0`. The parameter controls how *__get_free_pages* is invoked (see the section "get_free_page and Friends" in Chapter 8). The zero-order limitation (which forces pages to be allocated one at a time, rather than in larger groups) is dictated by the internals of *__get_free_pages*, the allocation function used by *scullp*. To maximize allocation performance, the Linux kernel maintains a list of free pages for each allocation order, and only the reference count of the first page in a cluster is incremented by *get_free_pages* and decremented by *free_pages*. The *mmap* method is disabled for a *scullp* device if the allocation order is greater than zero, because *nopage* deals with single pages rather than clusters of pages. *scullp*

映射位于地址64 kb的物理页面 - 规定，我们看到一个满是零的页面（本示例中的主机计算机是PC，但在其他平台上的结果将相同）：

```
morgana.root# ./mapper /dev/mem 0x10000 0x1000 | od -Ax -t x1
mapped "/dev/mem" from 65536 to 69632
000000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
*
001000
```

*remap_pfn_range*处理RAM的能力表明，诸如*scull*之类的基于内存的设备无法轻易实现*mmap*，因为其设备内存是召开的，而不是I/O内存。幸运的是，任何需要将RAM映射到用户空间的驱动程序都可以使用相对容易的解决方法。它使用了我们之前看到的*nopage*方法。

使用Nopage方法重新映射RAM

将真实RAM映射到用户空间的方法是使用vm_ops->nopage一次处理一个页面故障。示例实现是*scullp*模块的一部分，在第8章中引入。

scullp 是面向页面的炭设备。因为它是面向页面的，所以它可以在其内存上实现*mmap*。实现内存映射的代码使用“ Linux中的内存管理”部分中引入的一些概念。

在检查代码之前，让我们看一下*scullp*中影响*mmap*实现的设计选择：

- *scullp* 只要映射设备，就不会发布设备内存。这是政策的问题，而不是要求，它与*scull*和类似设备的行为不同，这些设备在打开以写作时被截断为0的长度。拒绝释放映射的*scullp*设备允许一个进程覆盖由另一个过程积极映射的区域，因此您可以测试并查看如何进行操作和设备内存相互作用。为避免释放映射的设备，驾驶员必须保留主动映射的数量；设备结构中的vmas字段用于此目的。

- 仅当*scullp* order参数（设置在Mod-Ule负载时间设置）为0时，才能执行内存映射。该参数控制如何调用*__get_free_pages*（请参见第8章中的“ get_free_page and friends”部分）。零级限制（该页面一次分配一个，而不是在较大的组中分配）由*__get_free_pages*的内部分配（v23}的内部分配）决定了*scullp*使用的分配函数。为了最大化分配性能，Linux内核维护每个分配顺序的免费页面列表，并且只有群集中的第一页的参考计数被*get_free_pages*递增，并由*free_pages*减少。如果分配顺序大于零，则禁用*mmap*方法，因为*nopage*处理单个页面而不是页面簇。 *scullp* simply does not know how to properly manage reference counts for pages that are part of higher-order allocations. (Return to the section "A scull Using Whole Pages: scullp" in Chapter 8 if you need a refresher on *scullp* and the memory allocation order value.)

The zero-order limitation is mostly intended to keep the code simple. It *is* possible to correctly implement *mmap* for multipage allocations by playing with the usage count of the pages, but it would only add to the complexity of the example without introducing any interesting information.

Code that is intended to map RAM according to the rules just outlined needs to implement the *open*, *close*, and *nopage* VMA methods; it also needs to access the memory map to adjust the page usage counts.

This implementation of *scullp_mmap* is very short, because it relies on the *nopage* function to do all the interesting work:

```
int scullp_mmap(struct file *filp, struct vm_area_struct *vma)
{
    struct inode *inode = filp->f_dentry->d_inode;

    /* refuse to map if order is not 0 */
    if (scullp_devices[iminor(inode)].order)
        return -ENODEV;

    /* don't do anything here: "nopage" will fill the holes */
    vma->vm_ops = &scullp_vm_ops;
    vma->vm_flags |= VM_RESERVED;
    vma->vm_private_data = filp->private_data;
    scullp_vma_open(vma);
    return 0;
}
```

The purpose of the `if` statement is to avoid mapping devices whose allocation order is not `0`. *scullp*'s operations are stored in the `vm_ops` field, and a pointer to the device structure is stashed in the `vm_private_data` field. At the end, `vm_ops->open` is called to update the count of active mappings for the device.

open and *close* simply keep track of the mapping count and are defined as follows:

```
void scullp_vma_open(struct vm_area_struct *vma)
{
    struct scullp_dev *dev = vma->vm_private_data;

    dev->vmas++;
}

void scullp_vma_close(struct vm_area_struct *vma)
{
    struct scullp_dev *dev = vma->vm_private_data;

    dev->vmas--;
}
```

> 根本不知道如何正确管理属于高阶分配一部分的页面的参考计数。（如果您需要在*scullp*和内存分配订单值上进行刷新，请返回第8章中的“使用整个页面：SCULLP”部分。）

零级限制主要是为了使代码保持简单。它可以通过播放页面的使用计数来正确实现*mmap*的乘数分配，但它只会添加示例的复杂性而无需引入任何有趣的信息。

旨在根据刚刚概述的规则映射RAM的代码需要实现*open*，*close*和*nopage* VMA方法；它还需要访问存储映射以调整页面使用计数。

*scullp_mmap*的实现非常短，因为它依赖*nopage*函数来完成所有有趣的工作：

```
int scullp_mmap(struct file *filp, struct vm_area_struct *vma)
{
    struct inode *inode = filp->f_dentry->d_inode;

    /* refuse to map if order is not 0 */
    if (scullp_devices[iminor(inode)].order)
        return -ENODEV;

    /* don't do anything here: "nopage" will fill the holes */
    vma->vm_ops = &scullp_vm_ops;
    vma->vm_flags |= VM_RESERVED;
    vma->vm_private_data = filp->private_data;
    scullp_vma_open(vma);
    return 0;
}
```

`if`语句的目的是避免映射分配顺序不是0的设备。*scullp*的操作存储在`vm_ops`字段中，并且将设备结构的指针藏在`vm_private_data`字段中。最后，`vm_ops->open`被调用以更新设备的活动映射计数。

*open*和*close*只需跟踪映射计数即可定义如下：

```
void scullp_vma_open(struct vm_area_struct *vma)
{
    struct scullp_dev *dev = vma->vm_private_data;

    dev->vmas++;
}

void scullp_vma_close(struct vm_area_struct *vma)
{
    struct scullp_dev *dev = vma->vm_private_data;

    dev->vmas--;
}
```

Most of the work is then performed by *nopage*. In the *scullp* implementation, the address parameter to *nopage* is used to calculate an offset into the device; the offset is then used to look up the correct page in the *scullp* memory tree:

```
struct page *scullp_vma_nopage(struct vm_area_struct *vma,
                                unsigned long address, int *type)
{
    unsigned long offset;
    struct scullp_dev *ptr, *dev = vma->vm_private_data;
    struct page *page = NOPAGE_SIGBUS;
    void *pageptr = NULL; /* default to "missing" */

    down(&dev->sem);
    offset = (address - vma->vm_start) + (vma->vm_pgoff << PAGE_SHIFT);
    if (offset >= dev->size) goto out; /* out of range */

    /*
     * Now retrieve the scullp device from the list,then the page.
     * If the device has holes, the process receives a SIGBUS when
     * accessing the hole.
     */
    offset >>= PAGE_SHIFT; /* offset is a number of pages */
    for (ptr = dev; ptr && offset >= dev->qset;) {
        ptr = ptr->next;
        offset -= dev->qset;
    }
    if (ptr && ptr->data) pageptr = ptr->data[offset];
    if (!pageptr) goto out; /* hole or end-of-file */
    page = virt_to_page(pageptr);

    /* got it, now increment the count */
    get_page(page);
    if (type)
        *type = VM_FAULT_MINOR;
  out:
    up(&dev->sem);
    return page;
}
```

scullp uses memory obtained with *get_free_pages*. That memory is addressed using logical addresses, so all *scullp_nopage* has to do to get a struct page pointer is to call *virt_to_page*.

The *scullp* device now works as expected, as you can see in this sample output from the *mapper* utility. Here, we send a directory listing of */dev* (which is long) to the *scullp* device and then use the *mapper* utility to look at pieces of that listing with *mmap*:

```
morgana% ls -l /dev > /dev/scullp
morgana% ./mapper /dev/scullp 0 140
mapped "/dev/scullp" from 0 (0x00000000) to 140 (0x0000008c)
total 232
crw-------    1 root     root      10,  10 Sep 15 07:40 adbmouse
```

然后，大部分工作由*nopage*执行。在*scullp*实现中，address参数to *nopage*用于计算设备中的偏置。然后，偏移量用于查找*scullp*内存树中的正确页面：

```
struct page *scullp_vma_nopage(struct vm_area_struct *vma,
                                unsigned long address, int *type)
{
    unsigned long offset;
    struct scullp_dev *ptr, *dev = vma->vm_private_data;
    struct page *page = NOPAGE_SIGBUS;
    void *pageptr = NULL; /* default to "missing" */

    down(&dev->sem);
    offset = (address - vma->vm_start) + (vma->vm_pgoff << PAGE_SHIFT);
    if (offset >= dev->size) goto out; /* out of range */

    /*
     * Now retrieve the scullp device from the list,then the page.
     * If the device has holes, the process receives a SIGBUS when
     * accessing the hole.
     */
    offset >>= PAGE_SHIFT; /* offset is a number of pages */
    for (ptr = dev; ptr && offset >= dev->qset;) {
        ptr = ptr->next;
        offset -= dev->qset;
    }
    if (ptr && ptr->data) pageptr = ptr->data[offset];
    if (!pageptr) goto out; /* hole or end-of-file */
    page = virt_to_page(pageptr);

    /* got it, now increment the count */
    get_page(page);
    if (type)
        *type = VM_FAULT_MINOR;
  out:
    up(&dev->sem);
    return page;
}
```

scullp 使用*get_free_pages*获得的内存。该内存是使用逻辑地址来解决的，因此所有*scullp_nopage*要获得struct page指针都必须呼叫*virt_to_page*。

*scullp*设备现在按预期工作，如您在*mapper*实用程序中的示例输出中所看到的那样。在这里，我们将*/dev* (的目录列表发送到)的)设备，然后使用*mapper*实用程序以*mmap*}：

```
morgana% ls -l /dev > /dev/scullp
morgana% ./mapper /dev/scullp 0 140
mapped "/dev/scullp" from 0 (0x00000000) to 140 (0x0000008c)
total 232
crw-------    1 root     root      10,  10 Sep 15 07:40 adbmouse
```

```
crw-r--r--    1 root     root      10, 175 Sep 15 07:40 agpgart
morgana% ./mapper /dev/scullp 8192 200
mapped "/dev/scullp" from 8192 (0x00002000) to 8392 (0x000020c8)
d0h1494
brw-rw----    1 root     floppy     2,  92 Sep 15 07:40 fd0h1660
brw-rw----    1 root     floppy     2,  20 Sep 15 07:40 fd0h360
brw-rw----    1 root     floppy     2,  12 Sep 15 07:40 fd0H360
```

Remapping Kernel Virtual Addresses

Although it's rarely necessary, it's interesting to see how a driver can map a kernel virtual address to user space using *mmap*. A true kernel virtual address, remember, is an address returned by a function such as *vmalloc*—that is, a virtual address mapped in the kernel page tables. The code in this section is taken from *scullv*, which is the module that works like *scullp* but allocates its storage through *vmalloc*.

Most of the *scullv* implementation is like the one we've just seen for *scullp*, except that there is no need to check the `order` parameter that controls memory allocation. The reason for this is that *vmalloc* allocates its pages one at a time, because single-page allocations are far more likely to succeed than multipage allocations. Therefore, the allocation order problem doesn't apply to *vmalloc*ed space.

Beyond that, there is only one difference between the *nopage* implementations used by *scullp* and *scullv*. Remember that *scullp*, once it found the page of interest, would obtain the corresponding `struct page` pointer with *virt_to_page*. That function does not work with kernel virtual addresses, however. Instead, you must use *vmalloc_to_page*. So the final part of the *scullv* version of *nopage* looks like:

```
  /*
   * After scullv lookup, "page" is now the address of the page
   * needed by the current process. Since it's a vmalloc address,
   * turn it into a struct page.
   */
  page = vmalloc_to_page(pageptr);

  /* got it, now increment the count */
  get_page(page);
  if (type)
      *type = VM_FAULT_MINOR;
out:
  up(&dev->sem);
  return page;
```

Based on this discussion, you might also want to map addresses returned by *ioremap* to user space. That would be a mistake, however; addresses from *ioremap* are special and cannot be treated like normal kernel virtual addresses. Instead, you should use *remap_pfn_range* to remap I/O memory areas into user space.

```
crw-r--r--    1 root     root      10, 175 Sep 15 07:40 agpgart
morgana% ./mapper /dev/scullp 8192 200
mapped "/dev/scullp" from 8192 (0x00002000) to 8392 (0x000020c8)
d0h1494
brw-rw----    1 root     floppy     2,  92 Sep 15 07:40 fd0h1660
brw-rw----    1 root     floppy     2,  20 Sep 15 07:40 fd0h360
brw-rw----    1 root     floppy     2,  12 Sep 15 07:40 fd0H360
```

重新映射内核虚拟地址

尽管很少有必要，但是有趣的是，如何使用*mmap*将驱动程序如何将内核虚拟地址映射到用户空间。请记住，真正的内核虚拟地址是由*vmalloc*之类的函数返回的地址，即在内核页面表中映射的虚拟地址。本节中的代码取自*scullv*，该模块的工作原理与*scullp*一样，但通过*vmalloc*分配其存储空间。

大多数*scullv*实现就像我们刚刚看到的*scullp*的实现一样，只是无需检查控制内存分配的order参数。这样做的原因是*vmalloc*一次分配第一页，因为单页分配比乘法分配更有可能成功。因此，分配顺序问题不适用于*vmalloc* ED空间。

除此之外，*scullp*和*scullv*使用的*nopage*实现之间只有一个区别。请记住，一旦找到感兴趣的页面，*scullp*将获得*virt_to_page*的相应struct page指针。但是，该功能与内核虚拟地址不起作用。相反，您必须使用*vmalloc_to_page*。因此，*nopage scullv*版本的最后一部分看起来像：

```
    /*
     * After scullv lookup, "page" is now the address of the page
     * needed by the current process. Since it's a vmalloc address,
     * turn it into a struct page.
     */
    page = vmalloc_to_page(pageptr);

    /* got it, now increment the count */
    get_page(page);
    if (type)
        *type = VM_FAULT_MINOR;
out:
    up(&dev->sem);
    return page;
```

基于此讨论，您可能还需要将*ioremap*返回的地址映射到用户空间。但是，那将是一个错误。 *ioremap*的地址很特殊，不能像普通内核虚拟地址一样对待。相反，您应该使用*remap_pfn_range*将I/O内存区域重建为用户空间。

Performing Direct I/O

Most I/O operations are buffered through the kernel. The use of a kernel-space buffer allows a degree of separation between user space and the actual device; this separation can make programming easier and can also yield performance benefits in many situations. There are cases, however, where it can be beneficial to perform I/O directly to or from a user-space buffer. If the amount of data being transferred is large, transferring data directly without an extra copy through kernel space can speed things up.

One example of direct I/O use in the 2.6 kernel is the SCSI tape driver. Streaming tapes can pass a lot of data through the system, and tape transfers are usually record-oriented, so there is little benefit to buffering data in the kernel. So, when the conditions are right (the user-space buffer is page-aligned, for example), the SCSI tape driver performs its I/O without copying the data.

That said, it is important to recognize that direct I/O does not always provide the performance boost that one might expect. The overhead of setting up direct I/O (which involves faulting in and pinning down the relevant user pages) can be significant, and the benefits of buffered I/O are lost. For example, the use of direct I/O requires that the *write* system call operate synchronously; otherwise the application does not know when it can reuse its I/O buffer. Stopping the application until each write completes can slow things down, which is why applications that use direct I/O often use asynchronous I/O operations as well.

The real moral of the story, in any case, is that implementing direct I/O in a char driver is usually unnecessary and can be hurtful. You should take that step only if you are sure that the overhead of buffered I/O is truly slowing things down. Note also that block and network drivers need not worry about implementing direct I/O at all; in both cases, higher-level code in the kernel sets up and makes use of direct I/O when it is indicated, and driver-level code need not even know that direct I/O is being performed.

The key to implementing direct I/O in the 2.6 kernel is a function called *get_user_pages*, which is declared in *<linux/mm.h>* with the following prototype:

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

执行直接I/O。

大多数I/O操作都通过内核进行缓冲。内核空间缓冲区的使用允许在用户空间和实际设备之间进行一定程度的分离；这种分离可以使编程变得更加容易，并且在许多情况下也可以产生绩效优势。但是，在某些情况下，直接对用户空间缓冲区或从用户空间缓冲区执行I/O可能是有益的。如果传输的数据量很大，则直接传输数据而没有通过内核空间额外的副本可以加快速度。

2.6内核中直接I/O使用的一个示例是SCSI磁带驱动程序。流磁带可以通过系统传递大量数据，并且磁带传输通常以记录为导向，因此在内核中缓冲数据几乎没有好处。因此，当条件正确时（例如，用户空间缓冲区分页符）时，SCSI磁带驱动程序在不复制数据的情况下执行其I/O。

就是说，重要的是要认识到直接I/O并不总是提供人们可能期望的绩效提升。设置直接I/O的开销（涉及故障并固定相关的用户页面）可能是明显的，并且丢失了缓冲I/O的好处。例如，直接I/O的使用要求*write*系统调用同步操作；否则，该应用程序将不知道何时可以重复使用其I/O缓冲区。停止应用程序直到每个写入完成都可以减慢速度，这就是为什么使用直接I/O的应用程序通常也使用异步I/O操作。

无论如何，故事的真正寓意是，在炭驾驶员中实施直接I/O通常是不必要的，可能会受到伤害。仅当您确定缓冲I/O的开销确实会减慢事情的速度时，才应该采取该步骤。还请注意，块和网络驱动程序不必担心完全实施直接I/O；在这两种情况下，内核中的高级代码均设置并在指示时使用直接I/O，并且驱动程序级代码甚至不必知道正在执行直接I/O。

在2.6内核中实现直接I/O的关键是一个称为*get_user_pages*的函数，该函数在<*linux/mm.h*>中声明，并具有以下原型：

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

This function has several arguments:

`tsk`
: A pointer to the task performing the I/O; its main purpose is to tell the kernel who should be charged for any page faults incurred while setting up the buffer. This argument is almost always passed as `current`.

`mm`
: A pointer to the memory management structure describing the address space to be mapped. The `mm_struct` structure is the piece that ties together all of the parts (VMAs) of a process's virtual address space. For driver use, this argument should always be `current->mm`.

`start`
`len`
: `start` is the (page-aligned) address of the user-space buffer, and `len` is the length of the buffer in pages.

`write`
`force`
: If `write` is nonzero, the pages are mapped for write access (implying, of course, that user space is performing a read operation). The `force` flag tells *get_user_pages* to override the protections on the given pages to provide the requested access; drivers should always pass `0` here.

`pages`
`vmas`
: Output parameters. Upon successful completion, `pages` contain a list of pointers to the `struct page` structures describing the user-space buffer, and `vmas` contains pointers to the associated VMAs. The parameters should, obviously, point to arrays capable of holding at least `len` pointers. Either parameter can be `NULL`, but you need, at least, the `struct page` pointers to actually operate on the buffer.

get_user_pages is a low-level memory management function, with a suitably complex interface. It also requires that the mmap reader/writer semaphore for the address space be obtained in read mode before the call. As a result, calls to *get_user_pages* usually look something like:

```
down_read(&current->mm->mmap_sem);
result = get_user_pages(current, current->mm, ...);
up_read(&current->mm->mmap_sem);
```

The return value is the number of pages actually mapped, which could be fewer than the number requested (but greater than zero).

Upon successful completion, the caller has a `pages` array pointing to the user-space buffer, which is locked into memory. To operate on the buffer directly, the kernel-space code must turn each `struct page` pointer into a kernel virtual address with *kmap* or *kmap_atomic*. Usually, however, devices for which direct I/O is justified are using DMA operations, so your driver will probably want to create a scatter/gather

此功能有几个参数:

`tsk`
: 指向执行I/O任务的指针；它的主要目的是告诉内核，在设置缓冲区时，应为任何页面故障收取费用。这个参数几乎总是以`current`的形式传递。

`mm` 指向内存管理结构的指针，描述要映射的地址空间。 `mm_struct`结构是将过程虚拟地址空间的所有部分（VMA）联系在一起的部分。为了使用驱动程序，此参数应始终为`current->mm`。

`start`
`len`
: `start` 是用户空间缓冲区的（页面对准）地址，`len`是页面中缓冲区的长度。

`write`
`force`
: 如果`write`非零，则映射页面以进行写入访问（当然，用户空间正在执行读取操作）。 `force`标志告诉*get_user_pages*覆盖给定页面上的保护措施，以提供请求的访问权限；驱动程序应始终在此处通过`0`。

`pages`
`vmas`
: 输出参数。成功完成后，`pages`包含描述用户空间缓冲区的`struct page`结构的指针列表，`vmas`包含相关VMA的指针。显然，这些参数应指向能够至少持有`len`指针的数组。任何一个参数都可以是`NULL`，但至少您需要`struct page`指针才能在缓冲区上实际操作。

get_user_pages 是一个低级内存管理功能，具有适当复杂的接口。它还要求在呼叫之前以读取模式获得MMAP读取器/作者信号量。结果，打电话给*get_user_pages*通常看起来像:

```
down_read(&current->mm->mmap_sem);
result = get_user_pages(current, current->mm, ...);
up_read(&current->mm->mmap_sem);
```

返回值是实际映射的页面数，可能少于请求的数字（但大于零）。

成功完成后，呼叫者的`pages`数组指向用户空间缓冲区，该缓冲区已锁定在内存中。要直接在缓冲区上操作，内核空间代码必须将每个`struct page`指针转换为带有*kmap*或*kmap_atomic*的内核虚拟地址。但是，通常，直接I/O的设备是合理的，因此使用DMA操作，因此您的驾驶员可能希望创建一个分散/收集

list from the array of `struct page` pointers. We discuss how to do this in the section, "Scatter/gather mappings."

Once your direct I/O operation is complete, you must release the user pages. Before doing so, however, you must inform the kernel if you changed the contents of those pages. Otherwise, the kernel may think that the pages are "clean," meaning that they match a copy found on the swap device, and free them without writing them out to backing store. So, if you have changed the pages (in response to a user-space read request), you must mark each affected page dirty with a call to:

```
void SetPageDirty(struct page *page);
```

(This macro is defined in *<linux/page-flags.h>*). Most code that performs this operation checks first to ensure that the page is not in the reserved part of the memory map, which is never swapped out. Therefore, the code usually looks like:

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

Since user-space memory is not normally marked reserved, this check should not strictly be necessary, but when you are getting your hands dirty deep within the memory management subsystem, it is best to be thorough and careful.

Regardless of whether the pages have been changed, they must be freed from the page cache, or they stay there forever. The call to use is:

```
void page_cache_release(struct page *page);
```

This call should, of course, be made *after* the page has been marked dirty, if need be.

Asynchronous I/O

One of the new features added to the 2.6 kernel was the *asynchronous I/O* capability. Asynchronous I/O allows user space to initiate operations without waiting for their completion; thus, an application can do other processing while its I/O is in flight. A complex, high-performance application can also use asynchronous I/O to have multiple operations going at the same time.

The implementation of asynchronous I/O is optional, and very few driver authors bother; most devices do not benefit from this capability. As we will see in the coming chapters, block and network drivers are fully asynchronous at all times, so only char drivers are candidates for explicit asynchronous I/O support. A char device can benefit from this support if there are good reasons for having more than one I/O operation outstanding at any given time. One good example is streaming tape drives, where the drive can stall and slow down significantly if I/O operations do not arrive quickly enough. An application trying to get the best performance out of a streaming drive could use asynchronous I/O to have multiple operations ready to go at any given time.

struct page指针数组中的列表。我们在“分散/收集映射”部分中讨论了如何执行此操作。

直接I/O操作完成后，必须发布用户页面。但是，在此之前，如果您更改了这些页面的内容，则必须通知内核。否则，内核可能会认为这些页面是“干净”的，这意味着它们匹配在交换设备上找到的副本，并在不将其写入备用商店的情况下释放它们。因此，如果您已更改页面（根据用户空间读取请求），则必须标记每个受影响的页面肮脏的呼叫：

```
void SetPageDirty(struct page *page);
```

(此宏定义在<*linux/page-flags.h*>)中。执行此操作的大多数代码首先检查，以确保页面不在存储映射的保留部分中，这从未换成。因此，代码通常看起来像：

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

由于通常没有标记的用户空间内存，因此不需要严格的检查，但是当您在内存管理子系统中深处弄脏时，最好要彻底和小心。

无论是否更改了页面，它们都必须从页面缓存中解放出来，否则它们会永远留在那里。使用的电话是：

```
void page_cache_release(struct page *page);
```

当然，应该进行此呼叫*after*页面已标记为脏（如果需要）。

异步I/O。

2.6内核中添加的新功能之一是*asynchronous I/O* capabil- ity。异步I/O允许用户空间启动操作而无需等待其完成；因此，应用程序在I/O处于飞行中时可以进行其他处理。复杂的高性能应用程序也可以使用异步I/O同时进行多个操作。

异步I/O的实现是可选的，很少有驾驶员作者烦恼。大多数设备无法从此功能中受益。正如我们将在章节中看到的那样，块和网络驱动程序始终是完全异步的，因此，只有char驱动程序才是明确异步I/O支持的候选人。如果有充分的理由在任何给定时间都有多个I/O操作，则炭设备可以从此支持中受益。一个很好的例子是流胶驱动器，如果I/O操作不够快，驱动器可以在其中停滞和放慢速度。试图从流媒体驱动器中获得最佳性能的应用程序可以使用异步I/O在任何给定时间进行多个操作。

For the rare driver author who needs to implement asynchronous I/O, we present a quick overview of how it works. We cover asynchronous I/O in this chapter, because its implementation almost always involves direct I/O operations as well (if you are buffering data in the kernel, you can usually implement asynchronous behavior without imposing the added complexity on user space).

Drivers supporting asynchronous I/O should include <*linux/aio.h*>. There are three *file_operations* methods for the implementation of asynchronous I/O:

```
ssize_t (*aio_read) (struct kiocb *iocb, char *buffer,
                     size_t count, loff_t offset);
ssize_t (*aio_write) (struct kiocb *iocb, const char *buffer,
                      size_t count, loff_t offset);
int (*aio_fsync) (struct kiocb *iocb, int datasync);
```

The *aio_fsync* operation is only of interest to filesystem code, so we do not discuss it further here. The other two, *aio_read* and *aio_write*, look very much like the regular *read* and *write* methods but with a couple of exceptions. One is that the `offset` parameter is passed by value; asynchronous operations never change the file position, so there is no reason to pass a pointer to it. These methods also take the `iocb` ("I/O control block") parameter, which we get to in a moment.

The purpose of the *aio_read* and *aio_write* methods is to initiate a read or write operation that may or may not be complete by the time they return. If it *is* possible to complete the operation immediately, the method should do so and return the usual status: the number of bytes transferred or a negative error code. Thus, if your driver has a *read* method called *my_read*, the following *aio_read* method is entirely correct (though rather pointless):

```
static ssize_t my_aio_read(struct kiocb *iocb, char *buffer,
                           ssize_t count, loff_t offset)
{
    return my_read(iocb->ki_filp, buffer, count, &offset);
}
```

Note that the `struct file` pointer is found in the `ki_filp` field of the `kiocb` structure.

If you support asynchronous I/O, you must be aware of the fact that the kernel can, on occasion, create "synchronous IOCBs." These are, essentially, asynchronous operations that must actually be executed synchronously. One may well wonder why things are done this way, but it's best to just do what the kernel asks. Synchronous operations are marked in the IOCB; your driver should query that status with:

```
int is_sync_kiocb(struct kiocb *iocb);
```

If this function returns a nonzero value, your driver must execute the operation synchronously.

In the end, however, the point of all this structure is to enable asynchronous operations. If your driver is able to initiate the operation (or, simply, to queue it until some future time when it can be executed), it must do two things: remember everything it

对于需要实施异步I/O的罕见驾驶员作者，我们可以快速概述其工作原理。我们在本章中介绍异步I/O，因为它的实现几乎始终涉及直接的I/O操作（如果您在内核中缓冲数据，通常可以实现异步行为，并且在用户空间上添加复杂性并将其施加了复杂性）。

支持异步I/O的驱动程序应包括<*linux/aio.h*>。有三种*file_operations*方法用于实现异步I/O：

```
ssize_t (*aio_read) (struct kiocb *iocb, char *buffer,
                     size_t count, loff_t offset);
ssize_t (*aio_write) (struct kiocb *iocb, const char *buffer,
                      size_t count, loff_t offset);
int (*aio_fsync) (struct kiocb *iocb, int datasync);
```

*aio_fsync*操作仅引起文件系统代码的兴趣，因此我们在这里不进一步讨论。另外两个，*aio_read*和*aio_write*，看起来非常类似于常规*read*和*write*方法，但有几个例外。一个是offset参数按值传递；异步操作永远不会更改文件位置，因此没有理由将指针传递给它。这些方法还采用iocb（“ i/o控制块”)参数，我们在片刻中获取。

*aio_read*和*aio_write*方法的目的是启动读取或写入操作，该操作可能会在它们返回之时完成，也可能不会完成。如果可以立即完成操作，则该方法应这样做并返回通常的状态：传输的字节数或负错误代码。因此，如果您的驱动程序具有称为*my_read*的*read*方法，则以下*aio_read*方法是完全正确的（尽管毫无意义）：

```
static ssize_t my_aio_read(struct kiocb *iocb, char *buffer,
                           ssize_t count, loff_t offset)
{
    return my_read(iocb->ki_filp, buffer, count, &offset);
}
```

请注意，struct file指针位于ki_filp kiocb结构的ki_filp字段中。

如果您支持异步I/O，则必须意识到，有时内核可以创建“同步IOCB”。从本质上讲，这些都是必须同步执行的异步操作。人们可能会想知道为什么要这样做，但是最好只做内核要求的事情。同步操作在IOCB中标记；您的驾驶员应以以下方式查询该状态

```
int is_sync_kiocb(struct kiocb *iocb);
```

如果此功能返回非零值，则您的驱动程序必须同步执行操作。

然而，最终，所有这些结构的重点是实现异步操作。如果您的驾驶员能够启动操作（或者简单地排队直到可以执行的时间），则必须做两件事：记住一切

needs to know about the operation, and return -EIOCBQUEUED to the caller. Remembering the operation information includes arranging access to the user-space buffer; once you return, you will not again have the opportunity to access that buffer while running in the context of the calling process. In general, that means you will likely have to set up a direct kernel mapping (with *get_user_pages*) or a DMA mapping. The -EIOCBQUEUED error code indicates that the operation is not yet complete, and its final status will be posted later.

When "later" comes, your driver must inform the kernel that the operation has completed. That is done with a call to *aio_complete*:

```
int aio_complete(struct kiocb *iocb, long res, long res2);
```

Here, iocb is the same IOCB that was initially passed to you, and res is the usual result status for the operation. res2 is a second result code that will be returned to user space; most asynchronous I/O implementations pass res2 as 0. Once you call *aio_complete*, you should not touch the IOCB or user buffer again.

An asynchronous I/O example

The page-oriented *scullp* driver in the example source implements asynchronous I/O. The implementation is simple, but it is enough to show how asynchronous operations should be structured.

The *aio_read* and *aio_write* methods don't actually do much:

```
static ssize_t scullp_aio_read(struct kiocb *iocb, char *buf, size_t count,
        loff_t pos)
{
    return scullp_defer_op(0, iocb, buf, count, pos);
}

static ssize_t scullp_aio_write(struct kiocb *iocb, const char *buf,
        size_t count, loff_t pos)
{
    return scullp_defer_op(1, iocb, (char *) buf, count, pos);
}
```

These methods simply call a common function:

```
struct async_work {
    struct kiocb *iocb;
    int result;
    struct work_struct work;
};

static int scullp_defer_op(int write, struct kiocb *iocb, char *buf,
        size_t count, loff_t pos)
{
    struct async_work *stuff;
    int result;
```

需要了解操作，然后将-EIOCBQUEUED返回到呼叫者。纪念操作信息包括安排对用户空间缓冲区的访问；返回后，在呼叫过程的上下文中运行时，您将无法再次访问该缓冲区。通常，这意味着您可能必须设置直接的内核映射（带有*get_user_pages*）或DMA映射。 -EIOCBQUEUED错误代码表示操作尚未完成，其最终状态将在稍后发布。

当“以后”到来时，您的驾驶员必须通知内核该操作已完成。这是通过呼叫*aio_complete*来完成的：

```
int aio_complete(struct kiocb *iocb, long res, long res2);
```

在这里，iocb是最初传递给您的IOCB，res是操作的通常结果状态。 res2是第二个结果代码，将返回用户空间；大多数异步I/O实现将0作为0通过。调用*aio_complete*后，您不应再次触摸IOCB或用户缓冲区。

异步I/O示例

示例源中的面向页面的*scullp*驱动程序实现异步I/O。实现很简单，但是足以说明如何将异步操作构建。

*aio_read*和*aio_write*方法实际上并没有做太多：

```
static ssize_t scullp_aio_read(struct kiocb *iocb, char *buf, size_t count,
        loff_t pos)
{
    return scullp_defer_op(0, iocb, buf, count, pos);
}

static ssize_t scullp_aio_write(struct kiocb *iocb, const char *buf,
        size_t count, loff_t pos)
{
    return scullp_defer_op(1, iocb, (char *) buf, count, pos);
}
```

这些方法只是调用一个常见的函数：

```
struct async_work {
    struct kiocb *iocb;
    int result;
    struct work_struct work;
};

static int scullp_defer_op(int write, struct kiocb *iocb, char *buf,
        size_t count, loff_t pos)
{
    struct async_work *stuff;
    int result;
```

```
        /* Copy now while we can access the buffer */
        if (write)
            result = scullp_write(iocb->ki_filp, buf, count, &pos);
        else
            result = scullp_read(iocb->ki_filp, buf, count, &pos);

        /* If this is a synchronous IOCB, we return our status now. */
        if (is_sync_kiocb(iocb))
            return result;

        /* Otherwise defer the completion for a few milliseconds. */
        stuff = kmalloc (sizeof (*stuff), GFP_KERNEL);
        if (stuff == NULL)
            return result; /* No memory, just complete now */
        stuff->iocb = iocb;
        stuff->result = result;
        INIT_WORK(&stuff->work, scullp_do_deferred_op, stuff);
        schedule_delayed_work(&stuff->work, HZ/100);
        return -EIOCBQUEUED;
    }
```

A more complete implementation would use *get_user_pages* to map the user buffer into kernel space. We chose to keep life simple by just copying over the data at the outset. Then a call is made to *is_sync_kiocb* to see if this operation must be completed synchronously; if so, the result status is returned, and we are done. Otherwise we remember the relevant information in a little structure, arrange for "completion" via a workqueue, and return `-EIOCBQUEUED`. At this point, control returns to user space.

Later on, the workqueue executes our completion function:

```
    static void scullp_do_deferred_op(void *p)
    {
        struct async_work *stuff = (struct async_work *) p;
        aio_complete(stuff->iocb, stuff->result, 0);
        kfree(stuff);
    }
```

Here, it is simply a matter of calling *aio_complete* with our saved information. A real driver's asynchronous I/O implementation is somewhat more complicated, of course, but it follows this sort of structure.

Direct Memory Access

Direct memory access, or DMA, is the advanced topic that completes our overview of memory issues. 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. Use of this mechanism can greatly increase throughput to and from a device, because a great deal of computational overhead is eliminated.

```
        /* Copy now while we can access the buffer */
        if (write)
            result = scullp_write(iocb->ki_filp, buf, count, &pos);
        else
            result = scullp_read(iocb->ki_filp, buf, count, &pos);

        /* If this is a synchronous IOCB, we return our status now. */
        if (is_sync_kiocb(iocb))
            return result;

        /* Otherwise defer the completion for a few milliseconds. */
        stuff = kmalloc (sizeof (*stuff), GFP_KERNEL);
        if (stuff == NULL)
            return result; /* No memory, just complete now */
        stuff->iocb = iocb;
        stuff->result = result;
        INIT_WORK(&stuff->work, scullp_do_deferred_op, stuff);
        schedule_delayed_work(&stuff->work, HZ/100);
        return -EIOCBQUEUED;
    }
```

更完整的实现将使用*get_user_pages*将用户缓冲区映射到内核空间中。我们选择通过一开始就复制数据来保持生活简单。然后呼叫*is_sync_kiocb*，以查看是否必须同步对此操作进行组合；如果是这样，结果状态将返回，我们完成了。否则，我们记得一些小结构中的相关信息，安排通过工作场所“完成”，然后返回-EIOCBQUEUED。此时，控制返回用户空间。

稍后，工作等执行我们的完成函数：

```
static void scullp_do_deferred_op(void *p)
{
    struct async_work *stuff = (struct async_work *) p;
    aio_complete(stuff->iocb, stuff->result, 0);
    kfree(stuff);
}
```

在这里，这只是用我们保存的信息调用*aio_complete*的问题。当然，真正的驱动程序异步I/O实现更为复杂，但是它遵循这种结构。

直接内存访问

直接内存访问或DMA是我们完成内存问题概述的高级主题。 DMA是硬件机制，允许外围组合可以直接传输其I/O数据，而无需涉及系统处理器。这种机制的使用可以大大增加到设备，因为消除了大量的计算开销。

Overview of a DMA Data Transfer

Before introducing the programming details, let's review how a DMA transfer takes place, considering only input transfers to simplify the discussion.

Data transfer can be triggered in two ways: either the software asks for data (via a function such as *read*) or the hardware asynchronously pushes data to the system.

In the first case, the steps involved can be summarized as follows:

1. When a process calls *read*, the driver method allocates a DMA buffer and instructs the hardware to transfer its data into that buffer. The process is put to sleep.
2. The hardware writes data to the DMA buffer and raises an interrupt when it's done.
3. The interrupt handler gets the input data, acknowledges the interrupt, and awakens the process, which is now able to read data.

The second case comes about when DMA is used asynchronously. This happens, for example, with data acquisition devices that go on pushing data even if nobody is reading them. In this case, the driver should maintain a buffer so that a subsequent *read* call will return all the accumulated data to user space. The steps involved in this kind of transfer are slightly different:

1. The hardware raises an interrupt to announce that new data has arrived.
2. The interrupt handler allocates a buffer and tells the hardware where to transfer its data.
3. The peripheral device writes the data to the buffer and raises another interrupt when it's done.
4. The handler dispatches the new data, wakes any relevant process, and takes care of housekeeping.

A variant of the asynchronous approach is often seen with network cards. These cards often expect to see a circular buffer (often called a *DMA ring buffer*) established in memory shared with the processor; each incoming packet is placed in the next available buffer in the ring, and an interrupt is signaled. The driver then passes the network packets to the rest of the kernel and places a new DMA buffer in the ring.

The processing steps in all of these cases emphasize that efficient DMA handling relies on interrupt reporting. While it is possible to implement DMA with a polling driver, it wouldn't make sense, because a polling driver would waste the performance benefits that DMA offers over the easier processor-driven I/O.*

* There are, of course, exceptions to everything; see the section "Receive Interrupt Mitigation" in Chapter 17 for a demonstration of how high-performance network drivers are best implemented using polling.

DMA数据传输的概述

在介绍编程详细信息之前，让我们回顾一下DMA转移是如何进行的，仅考虑输入转移以简化讨论。

数据传输可以通过两种方式触发：该软件要么询问数据（通过*read*）等函数，或者是硬件异步将数据推向系统。

在第一种情况下，涉及的步骤可以总结如下：

1。当一个进程调用*read*时，驱动程序方法会分配DMA缓冲区，并指示硬件将其数据传输到该缓冲区中。该过程入睡。 2。硬件将数据写入DMA缓冲区，并在完成后增加中断。 3。中断处理程序获取输入数据，确认中断并唤醒该过程，该过程现在可以读取数据。

第二种情况是在异步使用DMA时出现的。例如，即使没有人读取数据，也可以使用数据采集设备来推动数据。在这种情况下，驱动程序应维护缓冲区，以便随后的*read*调用将将所有累积数据返回到用户空间。这种转移所涉及的步骤略有不同：

1。硬件增加了一个中断，以宣布新数据已经到来。 2。中断处理程序分配一个缓冲区，并告诉硬件在哪里传输数据。 3。外围设备将数据写入缓冲区，并在完成后将另一个中断。 4。处理程序派遣新数据，唤醒任何相关的过程，并照顾家政服务。

与网卡通常可以看到异步方法的一种变体。这些卡通常希望看到与处理器共享的记忆中的圆形缓冲区（通常称为*DMA ring buffer*）；每个输入数据包都放在环中的下一个可用缓冲区中，并发出中断。然后，驾驶员将网络数据包传递到其余内核，并在环中放置一个新的DMA缓冲区。

在所有这些情况下，处理步骤都强调有效的DMA处理依赖于中断报告。虽然可以通过投票驱动程序实现DMA，但这是没有意义的，因为投票驱动程序会浪费DMA对更轻松的处理器驱动的I/O。

* There are, of course, exceptions to everything; see the section “Receive Interrupt Mitigation” in Chapter 17 for a demonstration of how high-performance network drivers are best implemented using polling.

Another relevant item introduced here is the DMA buffer. DMA requires device drivers to allocate one or more special buffers suited to DMA. Note that many drivers allocate their buffers at initialization time and use them until shutdown—the word *allocate* in the previous lists, therefore, means "get hold of a previously allocated buffer."

Allocating the DMA Buffer

This section covers the allocation of DMA buffers at a low level; we introduce a higher-level interface shortly, but it is still a good idea to understand the material presented here.

The main issue that arrises with DMA buffers is that, when they are bigger than one page, they must occupy contiguous pages in physical memory because the device transfers data using the ISA or PCI system bus, both of which carry physical addresses. It's interesting to note that this constraint doesn't apply to the SBus (see the section "SBus" in Chapter 12), which uses virtual addresses on the peripheral bus. Some architectures *can* also use virtual addresses on the PCI bus, but a portable driver cannot count on that capability.

Although DMA buffers can be allocated either at system boot or at runtime, modules can allocate their buffers only at runtime. (Chapter 8 introduced these techniques; the section "Obtaining Large Buffers" covered allocation at system boot, while "The Real Story of kmalloc" and "get_free_page and Friends" described allocation at runtime.) Driver writers must take care to allocate the right kind of memory when it is used for DMA operations; not all memory zones are suitable. In particular, high memory may not work for DMA on some systems and with some devices—the peripherals simply cannot work with addresses that high.

Most devices on modern buses can handle 32-bit addresses, meaning that normal memory allocations work just fine for them. Some PCI devices, however, fail to implement the full PCI standard and cannot work with 32-bit addresses. And ISA devices, of course, are limited to 24-bit addresses only.

For devices with this kind of limitation, memory should be allocated from the DMA zone by adding the `GFP_DMA` flag to the *kmalloc* or *get_free_pages* call. When this flag is present, only memory that can be addressed with 24 bits is allocated. Alternatively, you can use the generic DMA layer (which we discuss shortly) to allocate buffers that work around your device's limitations.

Do-it-yourself allocation

We have seen how *get_free_pages* can allocate up to a few megabytes (as order can range up to `MAX_ORDER`, currently 11), but high-order requests are prone to fail even

这里介绍的另一个相关项目是DMA缓冲区。 DMA要求设备传动器分配一个或多个适合DMA的特殊缓冲区。请注意，许多驱动程序在初始化时分配了缓冲区，并使用它们直到关机为止 - 因此，上一个列表中的单词*allocate*单词意味着“握住先前分配的缓冲区”。

分配DMA缓冲区

本节涵盖了低水平的DMA缓冲区的分配；我们很快介绍了一个高级界面，但是了解此处介绍的材料仍然是一个好主意。

带有DMA缓冲区的主要问题是，当它们大于一页时，它们必须在物理内存中占据连续页面，因为该设备使用ISA或PCI系统总线传输数据，这两者都带有物理地址。有趣的是，此约束不适用于SBU（请参阅第12章中的“ SBU”部分，该部分使用外围总线上的虚拟地址。某些架构*can*还使用PCI总线上的虚拟地址，但是便携式驱动程序无法指望该功能。

尽管可以在系统启动时或运行时分配DMA缓冲区，但Mod-Ules只能在运行时分配其缓冲区。（第8章介绍了这些技术；“获得大型缓冲区”部分涵盖了系统靴的分配，而“ kmalloc的真实故事”和“ get_free_page and Friends and Friends and Friends”描述了运行时的分配。）驾驶员作家必须小心去分配适用于DMA操作的正确记忆；并非所有记忆区都合适。在特殊情况下，高内存可能对某些系统和某些设备不适用于DMA，外围设备根本无法与高地址一起使用。

现代公交车上的大多数设备都可以处理32位地址，这意味着正常的内存分配对他们来说很好。但是，一些PCI设备无法实现完整的PCI标准，并且无法与32位地址一起使用。当然，ISA设备仅限于24位地址。

对于具有此类限制的设备，应通过将`GFP_DMA`标志添加到*kmalloc*或*get_free_pages*调用中，从DMA区域分配内存。当存在此标志时，只能分配24位可以解决的内存。替代性，您可以使用通用DMA层（我们很快讨论）来分配围绕设备限制的buffers。

做自己动手分配

我们已经看到*get_free_pages*如何分配几兆字节（因为订单可以范围为`MAX_ORDER`，目前为11），但是高阶请求甚至容易失败

when the requested buffer is far less than 128 KB, because system memory becomes fragmented over time.*

When the kernel cannot return the requested amount of memory or when you need more than 128 KB (a common requirement for PCI frame grabbers, for example), an alternative to returning -ENOMEM is to allocate memory at boot time or reserve the top of physical RAM for your buffer. We described allocation at boot time in the section "Obtaining Large Buffers" in Chapter 8, but it is not available to modules. Reserving the top of RAM is accomplished by passing a mem= argument to the kernel at boot time. For example, if you have 256 MB, the argument mem=255M keeps the kernel from using the top megabyte. Your module could later use the following code to gain access to such memory:

```
dmabuf = ioremap (0xFF00000 /* 255M */, 0x100000 /* 1M */);
```

The *allocator*, part of the sample code accompanying the book, offers a simple API to probe and manage such reserved RAM and has been used successfully on several architectures. However, this trick doesn't work when you have an high-memory system (i.e., one with more physical memory than could fit in the CPU address space).

Another option, of course, is to allocate your buffer with the GFP_NOFAIL allocation flag. This approach does, however, severely stress the memory management subsystem, and it runs the risk of locking up the system altogether; it is best avoided unless there is truly no other way.

If you are going to such lengths to allocate a large DMA buffer, however, it is worth putting some thought into alternatives. If your device can do scatter/gather I/O, you can allocate your buffer in smaller pieces and let the device do the rest. Scatter/gather I/O can also be used when performing direct I/O into user space, which may well be the best solution when a truly huge buffer is required.

Bus Addresses

A device driver using DMA has to talk to hardware connected to the interface bus, which uses physical addresses, whereas program code uses virtual addresses.

As a matter of fact, the situation is slightly more complicated than that. DMA-based hardware uses *bus*, rather than *physical*, addresses. Although ISA and PCI bus addresses are simply physical addresses on the PC, this is not true for every platform. Sometimes the interface bus is connected through bridge circuitry that maps I/O addresses to different physical addresses. Some systems even have a page-mapping scheme that can make arbitrary pages appear contiguous to the peripheral bus.

* The word *fragmentation* is usually applied to disks to express the idea that files are not stored consecutively on the magnetic medium. The same concept applies to memory, where each virtual address space gets scattered throughout physical RAM, and it becomes difficult to retrieve consecutive free pages when a DMA buffer is requested.

当请求的缓冲区远小于128 kb时，由于系统内存会随着时间而分散。[*]

当内核无法返回所需的内存量或需要超过128 kb（例如，PCI框架抓手的常见要求）时，返回-ENOMEM的替代方案是在启动时间分配内存或保留用于缓冲区的物理RAM的顶部。我们在第8章中的“获取大型缓冲区”部分中的启动时间分配了分配，但是模块不可用。保留RAM顶部是通过在启动时向内核传递mem=参数来完成的。例如，如果您有256 MB，则参数mem=255M可以防止内核使用顶部的兆字节。您的模块以后可以使用以下代码访问此类内存：

```
dmabuf = ioremap (0xFF00000 /* 255M */, 0x100000 /* 1M */);
```

*allocator*是本书附带示例代码的一部分，提供了一个简单的API来探测和管理此类保留的RAM，并已成功地用于几个架构。但是，当您拥有高内存系统时（即具有比CPU地址空间中的更多物理内存）时，此技巧无效。

当然，另一个选择是用GFP_NOFAIL分配标志分配缓冲区。但是，这种方法确实严重强调了内存管理子系统，并且它有完全锁定系统的风险。最好避免使用它，除非确实没有其他方法。

但是，如果您要竭尽全力分配大型DMA缓冲液，那么值得将一些思想置于替代方案中。如果您的设备可以进行散射/收集I/O，则可以将缓冲区分配成较小的零件，并让设备进行其余的操作。当将I/O执行直接I/O进入用户空间时，也可以使用散点/收集I/O，当需要真正的巨大缓冲区时，这很可能是最好的解决方案。

巴士地址

使用DMA的设备驱动程序必须与连接到接口总线的硬件进行对话，该界面总线使用物理地址，而程序代码使用虚拟地址。

事实上，情况比这复杂得多。基于DMA的硬件使用*bus*，而不是*physical*，地址。尽管ISA和PCI总线地址只是PC上的物理地址，但对于每个平台并非如此。有时，接口总线是通过桥梁电路连接的，将I/O地址映射到不同的物理地址。有些系统甚至具有一个可以使任意页面的页面映射方案与外围总线看起来连续。

[*] The word *fragmentation* is usually applied to disks to express the idea that files are not stored consecutively on the magnetic medium. The same concept applies to memory, where each virtual address space gets scattered throughout physical RAM, and it becomes difficult to retrieve consecutive free pages when a DMA buffer is requested.

At the lowest level (again, we'll look at a higher-level solution shortly), the Linux kernel provides a portable solution by exporting the following functions, defined in <*asm/io.h*>. The use of these functions is strongly discouraged, because they work properly only on systems with a very simple I/O architecture; nonetheless, you may encounter them when working with kernel code.

```
unsigned long virt_to_bus(volatile void *address);
void *bus_to_virt(unsigned long address);
```

These functions perform a simple conversion between kernel logical addresses and bus addresses. They do not work in any situation where an I/O memory management unit must be programmed or where bounce buffers must be used. The right way of performing this conversion is with the generic DMA layer, so we now move on to that topic.

The Generic DMA Layer

DMA operations, in the end, come down to allocating a buffer and passing bus addresses to your device. However, the task of writing portable drivers that perform DMA safely and correctly on all architectures is harder than one might think. Different systems have different ideas of how cache coherency should work; if you do not handle this issue correctly, your driver may corrupt memory. Some systems have complicated bus hardware that can make the DMA task easier—or harder. And not all systems can perform DMA out of all parts of memory. Fortunately, the kernel provides a bus- and architecture-independent DMA layer that hides most of these issues from the driver author. We strongly encourage you to use this layer for DMA operations in any driver you write.

Many of the functions below require a pointer to a `struct device`. This structure is the low-level representation of a device within the Linux device model. It is not something that drivers often have to work with directly, but you do need it when using the generic DMA layer. Usually, you can find this structure buried inside the bus specific that describes your device. For example, it can be found as the `dev` field in `struct pci_device` or `struct usb_device`. The `device` structure is covered in detail in Chapter 14.

Drivers that use the following functions should include <*linux/dma-mapping.h*>.

Dealing with difficult hardware

The first question that must be answered before attempting DMA is whether the given device is capable of such an operation on the current host. Many devices are limited in the range of memory they can address, for a number of reasons. By default, the kernel assumes that your device can perform DMA to any 32-bit address. If this is not the case, you should inform the kernel of that fact with a call to:

```
int dma_set_mask(struct device *dev, u64 mask);
```

在最低级别（同样，我们将尽快查看更高级别的解决方案），Linux kernel通过导出以下函数（V2}中定义的以下功能，提供便携式解决方案。这些功能的使用强烈灰心，因为它们仅在具有非常简单的I/O架构的系统上正常工作；但是，在使用内核代码时，您可能会遇到它们。

```
unsigned long virt_to_bus(volatile void *address);
void *bus_to_virt(unsigned long address);
```

这些功能在内核逻辑地址和总线地址之间执行简单的转换。它们在必须对I/O内存管理单元进行编程或必须使用弹跳缓冲区的任何情况下工作。执行此转换的正确方法是使用通用DMA层，因此我们现在继续使用该主题。

通用DMA层

最终，DMA操作归结为将缓冲区和将总线地址分配给您的设备。但是，编写安全，正确执行DMA的便携式驱动程序的任务比人们想象的要难。不同的系统对缓存相干性的工作方式有不同的想法。如果您无法正确处理此问题，则驾驶员可能会损坏内存。一些系统具有复杂的总线硬件，可以使DMA任务更容易或更难。并非所有系统都可以从内存的所有部分中执行DMA。幸运的是，内核提供了与公交和建筑无关的DMA层，该层将大多数问题隐藏在驾驶员作者身上。我们强烈建议您将此层用于您编写的任何驱动程序中的DMA操作。

以下许多功能都需要指向`struct device`的指针。该结构是Linux设备模型中设备的低级表示。这不是驱动程序通常必须直接使用的东西，但是在使用通用DMA层时，您确实需要它。通常，您可以找到埋在总线内的特定于描述您设备的结构。例如，可以在`struct pci_device`或`struct usb_device`中的`dev`字段中找到它。 `device`结构在第14章中详细介绍。

使用以下功能的驱动程序应包括<*linux/dma-mapping.h*>。

处理困难的硬件

尝试DMA之前必须回答的第一个问题是，给定设备是否能够在当前主机上进行此类操作。由于多种原因，许多设备在可以解决的内存范围内受到限制。默认情况下，内核假设您的设备可以执行DMA到任何32位地址。如果不是这种情况，则应通过呼吁：

```
int dma_set_mask(struct device *dev, u64 mask);
```

The mask should show the bits that your device can address; if it is limited to 24 bits, for example, you would pass mask as 0x0FFFFFF. The return value is nonzero if DMA is possible with the given mask; if *dma_set_mask* returns 0, you are not able to use DMA operations with this device. Thus, the initialization code in a driver for a device limited to 24-bit DMA operations might look like:

```
if (dma_set_mask (dev, 0xffffff))
    card->use_dma = 1;
else {
    card->use_dma = 0;   /* We'll have to live without DMA */
    printk (KERN_WARN, "mydev: DMA not supported\n");
}
```

Again, if your device supports normal, 32-bit DMA operations, there is no need to call *dma_set_mask*.

DMA mappings

A *DMA mapping* is a combination of allocating a DMA buffer and generating an address for that buffer that is accessible by the device. It is tempting to get that address with a simple call to *virt_to_bus*, but there are strong reasons for avoiding that approach. The first of those is that reasonable hardware comes with an IOMMU that provides a set of *mapping registers* for the bus. The IOMMU can arrange for any physical memory to appear within the address range accessible by the device, and it can cause physically scattered buffers to look contiguous to the device. Making use of the IOMMU requires using the generic DMA layer; *virt_to_bus* is not up to the task.

Note that not all architectures have an IOMMU; in particular, the popular x86 platform has no IOMMU support. A properly written driver need not be aware of the I/O support hardware it is running over, however.

Setting up a useful address for the device may also, in some cases, require the establishment of a *bounce buffer*. Bounce buffers are created when a driver attempts to perform DMA on an address that is not reachable by the peripheral device—a high-memory address, for example. Data is then copied to and from the bounce buffer as needed. Needless to say, use of bounce buffers can slow things down, but sometimes there is no alternative.

DMA mappings must also address the issue of cache coherency. Remember that modern processors keep copies of recently accessed memory areas in a fast, local cache; without this cache, reasonable performance is not possible. If your device changes an area of main memory, it is imperative that any processor caches covering that area be invalidated; otherwise the processor may work with an incorrect image of main memory, and data corruption results. Similarly, when your device uses DMA to read data from main memory, any changes to that memory residing in processor caches must be flushed out first. These *cache coherency* issues can create no end of obscure and difficult-to-find bugs if the programmer is not careful. Some architectures manage cache mask应显示您的设备可以地址的位；例如，如果仅限于24位，则将mask作为0x0FFFFFF传递。如果给定的mask可能可以使用DMA，则返回值为非零;如果*dma_set_mask*返回0，则无法使用此设备使用DMA操作。因此，设备限制为24位DMA操作的驱动程序中的初始化代码可能看起来像:

```
if (dma_set_mask (dev, 0xffffff))
    card->use_dma = 1;
else {
    card->use_dma = 0;   /* We'll have to live without DMA */
    printk (KERN_WARN, "mydev: DMA not supported\n");
}
```

同样，如果您的设备支持正常的32位DMA操作，则无需致电*dma_set_mask*。

DMA映射

*DMA mapping*是分配DMA缓冲区并生成该缓冲区可通过设备访问的地址的组合。用简单的呼叫*virt_to_bus*来获取该地址很诱人，但是有很大的理由避免这种方法。第一个是合理的硬件带有一个IOMMU，该硬件为总线提供了一组*mapping registers*。 IOMMU可以安排任何物理内存出现在设备上可访问的地址范围内，并且可能导致物理散射的缓冲区与设备看起来连续。利用IOMMU需要使用通用DMA层； *virt_to_bus*不符合任务。

请注意，并非所有的架构都有IOMMU；特别是，流行的X86平台没有IOMMU支持。但是，编写正确的驱动程序不必知道它正在运行的I/O支持硬件。

在某些情况下，为设备设置有用的地址也可能需要*bounce buffer*的建立。当驱动程序尝试在外围设备无法触及的地址（例如高内存地址）上执行DMA时，会创建弹跳缓冲区。然后根据需要将数据复制到弹跳缓冲区。不用说，使用弹跳缓冲器可以放慢速度，但是有时别无选择。

DMA映射还必须解决缓存相干性的问题。请记住，MODERN处理器在快速的本地缓存中保留最近访问的内存区域的副本；没有此缓存，就不可能进行合理的性能。如果您的设备更改了主内存的区域，则必须将覆盖该区域的任何处理器缓存无效；否则，处理器可能会与主要成员的不正确图像一起工作，并且数据损坏结果。同样，当您的设备使用DMA从主内存中读取数据时，必须先将驻留在处理器缓存中的内存中的任何更改。如果程序员不小心，这些*cache coherency*问题不会造成晦涩和困难的错误。一些架构管理缓存

coherency in the hardware, but others require software support. The generic DMA layer goes to great lengths to ensure that things work correctly on all architectures, but, as we will see, proper behavior requires adherence to a small set of rules.

The DMA mapping sets up a new type, `dma_addr_t`, to represent bus addresses. Variables of type `dma_addr_t` should be treated as opaque by the driver; the only allowable operations are to pass them to the DMA support routines and to the device itself. As a bus address, `dma_addr_t` may lead to unexpected problems if used directly by the CPU.

The PCI code distinguishes between two types of DMA mappings, depending on how long the DMA buffer is expected to stay around:

Coherent DMA mappings
: These mappings usually exist for the life of the driver. A coherent buffer must be simultaneously available to both the CPU and the peripheral (other types of mappings, as we will see later, can be available only to one or the other at any given time). As a result, coherent mappings must live in cache-coherent memory. Coherent mappings can be expensive to set up and use.

Streaming DMA mappings
: Streaming mappings are usually set up for a single operation. Some architectures allow for significant optimizations when streaming mappings are used, as we see, but these mappings also are subject to a stricter set of rules in how they may be accessed. The kernel developers recommend the use of streaming mappings over coherent mappings whenever possible. There are two reasons for this recommendation. The first is that, on systems that support mapping registers, each DMA mapping uses one or more of them on the bus. Coherent mappings, which have a long lifetime, can monopolize these registers for a long time, even when they are not being used. The other reason is that, on some hardware, streaming mappings can be optimized in ways that are not available to coherent mappings.

The two mapping types must be manipulated in different ways; it's time to look at the details.

Setting up coherent DMA mappings

A driver can set up a coherent mapping with a call to *dma_alloc_coherent*:

```
void *dma_alloc_coherent(struct device *dev, size_t size,
                         dma_addr_t *dma_handle, int flag);
```

This function handles both the allocation and the mapping of the buffer. The first two arguments are the device structure and the size of the buffer needed. The function returns the result of the DMA mapping in two places. The return value from the function is a kernel virtual address for the buffer, which may be used by the driver; the associated bus address, meanwhile, is returned in `dma_handle`. Allocation is handled in

硬件的相干性，但其他需要软件支持。通用DMA层的长度很长，以确保事物在所有体系结构上都正确起作用，但是，正如我们将看到的那样，正确的行为需要遵守一小部分规则。

DMA映射设置了一种新类型dma_addr_t来表示总线地址。类型dma_addr_t的变量应由驱动程序视为不透明；唯一的允许操作是将它们传递到DMA支持例程和设备本身。作为总线地址，如果CPU直接使用，dma_addr_t可能会导致意外问题。

PCI代码区分了两种类型的DMA映射，具体取决于预期DMA缓冲区的时间：

Coherent DMA mappings
: 这些映射通常存在于驾驶员的生活中。 CPU和外围都必须同时使用连贯的缓冲区（正如我们稍后将看到的其他类型的映射，在任何给定时间只能向一个或另一个可用）。结果，连贯的映射必须生活在高速缓存的内存中。建立和使用的连贯映射可能很昂贵。

Streaming DMA mappings
: 通常为单个操作设置流映射。如我们所见，在使用流映射时，一些架构允许进行明显的优化，但是这些映射也需要访问如何访问的规则。内核开发人员建议尽可能在相干映射上使用流映射。此建议有两个原因。首先是，在支持映射寄存器的系统上，每个DMA映射在总线上使用其中一个或多个。一辈子很长的一致映射，即使不使用它们，也可以长期垄断这些寄存器。另一个原因是，在某些硬件上，可以以不可用的方式来优化流映射。

必须以不同的方式操纵这两种映射类型。是时候查看细节了。

设置连贯的DMA映射

驾驶员可以设置一个连贯的映射，并调用*dma_alloc_coherent*：

```
void *dma_alloc_coherent(struct device *dev, size_t size,
                         dma_addr_t *dma_handle, int flag);
```

此函数处理缓冲区的分配和映射。前两个参数是设备结构和所需的缓冲区的大小。该函数返回两个位置的DMA映射结果。来自功能的返回值是缓冲区的内核虚拟地址，驱动程序可以使用。同时，相关的总线地址在dma_handle中返回。分配已处理

this function so that the buffer is placed in a location that works with DMA; usually the memory is just allocated with *get_free_pages* (but note that the size is in bytes, rather than an order value). The `flag` argument is the usual `GFP_` value describing how the memory is to be allocated; it should usually be `GFP_KERNEL` (usually) or `GFP_ATOMIC` (when running in atomic context).

When the buffer is no longer needed (usually at module unload time), it should be returned to the system with *dma_free_coherent*:

```
void dma_free_coherent(struct device *dev, size_t size,
                        void *vaddr, dma_addr_t dma_handle);
```

Note that this function, like many of the generic DMA functions, requires that all of the size, CPU address, and bus address arguments be provided.

DMA pools

A *DMA pool* is an allocation mechanism for small, coherent DMA mappings. Mappings obtained from *dma_alloc_coherent* may have a minimum size of one page. If your device needs smaller DMA areas than that, you should probably be using a DMA pool. DMA pools are also useful in situations where you may be tempted to perform DMA to small areas embedded within a larger structure. Some very obscure driver bugs have been traced down to cache coherency problems with structure fields adjacent to small DMA areas. To avoid this problem, you should always allocate areas for DMA operations explicitly, away from other, non-DMA data structures.

The DMA pool functions are defined in <*linux/dmapool.h*>.

A DMA pool must be created before use with a call to:

```
struct dma_pool *dma_pool_create(const char *name, struct device *dev,
                                 size_t size, size_t align,
                                 size_t allocation);
```

Here, `name` is a name for the pool, `dev` is your device structure, `size` is the size of the buffers to be allocated from this pool, `align` is the required hardware alignment for allocations from the pool (expressed in bytes), and `allocation` is, if nonzero, a memory boundary that allocations should not exceed. If `allocation` is passed as 4096, for example, the buffers allocated from this pool do not cross 4-KB boundaries.

When you are done with a pool, it can be freed with:

```
void dma_pool_destroy(struct dma_pool *pool);
```

You should return all allocations to the pool before destroying it.

Allocations are handled with *dma_pool_alloc*:

```
void *dma_pool_alloc(struct dma_pool *pool, int mem_flags,
                     dma_addr_t *handle);
```

For this call, `mem_flags` is the usual set of `GFP_` allocation flags. If all goes well, a region of memory (of the size specified when the pool was created) is allocated and

此功能使缓冲区放置在与DMA一起使用的位置；通常，内存只是用 *get_free_pages* (分配，但请注意，大小为字节，而不是订单值)。 `flag`参数是通常的`GFP_`值，描述了如何分配内存；通常应为`GFP_KERNEL` (通常)或`GFP_ATOMIC` (在原子上下文)中运行时。

当不再需要缓冲区（通常在模块卸载时间）时，应将其返回到系统中 *dma_free_coherent*：

```
void dma_free_coherent(struct device *dev, size_t size,
                        void *vaddr, dma_addr_t dma_handle);
```

请注意，与许多通用DMA功能一样，此功能要求提供所有大小，CPU地址和总线地址参数。

DMA池

*DMA pool*是针对小型，连贯的DMA映射的分配机制。从*dma_alloc_coherent*获得的地图可能具有最小大小为一页。如果您的设备需要比这更小的DMA区域，则可能应该使用DMA池。 DMA池在您可能很想在嵌入较大结构内的小区域执行DMA的情况下也很有用。一些非常晦涩的驱动程序被追溯到与小型DMA区域相邻的结构字段的缓存相干问题。为了避免此问题，您应始终明确地分配DMA操作区域，远离其他非DMA数据结构。

DMA池函数在<*linux/dmapool.h*>中定义。

必须在使用呼叫之前创建DMA池：

```
struct dma_pool *dma_pool_create(const char *name, struct device *dev,
                                 size_t size, size_t align,
                                 size_t allocation);
```

Here, `name` is a name for the pool, `dev` is your device structure, `size` is the size of the buf fers to be allocated from this pool, `align` is the required hardware alignment for allocati ons from the pool (expressed in bytes), and `allocation` is, if nonzero, a mem- ory bound ary that allocations should not exceed.例如，如果将`allocation`作为4096传递，则从该池分配的缓冲区将不会跨越4 kb边界。

当您完成游泳池时，它可以释放：

```
void dma_pool_destroy(struct dma_pool *pool);
```

在销毁它之前，您应该将所有分配给游泳池。

分配使用*dma_pool_alloc*：

```
void *dma_pool_alloc(struct dma_pool *pool, int mem_flags,
                     dma_addr_t *handle);
```

对于此调用，`mem_flags`是`GFP_`分配标志的通常集。如果一切顺利，则分配了（创建池时指定的大小）的内存区域，然后分配

returned. As with *dma_alloc_coherent*, the address of the resulting DMA buffer is returned as a kernel virtual address and stored in `handle` as a bus address.

Unneeded buffers should be returned to the pool with:

```
void dma_pool_free(struct dma_pool *pool, void *vaddr, dma_addr_t addr);
```

Setting up streaming DMA mappings

Streaming mappings have a more complicated interface than the coherent variety, for a number of reasons. These mappings expect to work with a buffer that has already been allocated by the driver and, therefore, have to deal with addresses that they did not choose. On some architectures, streaming mappings can also have multiple, discontiguous pages and multipart "scatter/gather" buffers. For all of these reasons, streaming mappings have their own set of mapping functions.

When setting up a streaming mapping, you must tell the kernel in which direction the data is moving. Some symbols (of type `enum dma_data_direction`) have been defined for this purpose:

`DMA_TO_DEVICE`
`DMA_FROM_DEVICE`
: These two symbols should be reasonably self-explanatory. If data is being sent to the device (in response, perhaps, to a *write* system call), `DMA_TO_DEVICE` should be used; data going to the CPU, instead, is marked with `DMA_FROM_DEVICE`.

`DMA_BIDIRECTIONAL`
: If data can move in either direction, use `DMA_BIDIRECTIONAL`.

`DMA_NONE`
: This symbol is provided only as a debugging aid. Attempts to use buffers with this "direction" cause a kernel panic.

It may be tempting to just pick `DMA_BIDIRECTIONAL` at all times, but driver authors should resist that temptation. On some architectures, there is a performance penalty to pay for that choice.

When you have a single buffer to transfer, map it with *dma_map_single*:

```
dma_addr_t dma_map_single(struct device *dev, void *buffer, size_t size,
                          enum dma_data_direction direction);
```

The return value is the bus address that you can pass to the device or `NULL` if something goes wrong.

Once the transfer is complete, the mapping should be deleted with *dma_unmap_single*:

```
void dma_unmap_single(struct device *dev, dma_addr_t dma_addr, size_t size,
                      enum dma_data_direction direction);
```

Here, the `size` and `direction` arguments must match those used to map the buffer.

返回。与*dma_alloc_coherent*一样，所得DMA缓冲区的地址作为内核虚拟地址返回，并将其存储在handle中作为总线地址。

不需要的缓冲区应与：

```
void dma_pool_free(struct dma_pool *pool, void *vaddr, dma_addr_t addr);
```

设置流式DMA映射

由于多种原因，流映射的界面比连贯的品种更复杂。这些映射期望与驾驶员已经分配的缓冲区一起使用，因此必须处理他们没有选择的地址。在某些架构上，流映射还可以具有多个不连续的页面和多部分“散点/收集”缓冲区。由于所有这些原因，流映射具有自己的一套映射功能。

设置流映射时，必须告诉内核数据正在移动哪个方向。为此目的定义了一些（类型enum dma_data_direction）的符号：

DMA_TO_DEVICE
DMA_FROM_DEVICE
: 这两个符号应该是合理的自我解释。如果将数据发送到设备（也许是响应*write*系统调用），则应使用DMA_TO_DEVICE；取而代之的是DMA_FROM_DEVICE标记的数据。

DMA_BIDIRECTIONAL
: 如果数据可以朝任何方向移动，请使用DMA_BIDIRECTIONAL。

DMA_NONE
: 此符号仅作为调试援助提供。尝试使用此“方向”的缓冲区引起内核恐慌。

始终选择DMA_BIDIRECTIONAL可能很诱人，但是驾驶员作者应该抵制这种诱惑。在某些体系结构上，要为此选择付出绩效罚款。

当您有一个可以传输的单个缓冲区时，请用*dma_map_single*映射它：

```
dma_addr_t dma_map_single(struct device *dev, void *buffer, size_t size,
                          enum dma_data_direction direction);
```

返回值是您可以将其传递到设备或NULL的总线地址，如果某事会出错。

传输完成后，应使用*dma_unmap_single*删除映射：

```
void dma_unmap_single(struct device *dev, dma_addr_t dma_addr, size_t size,
                      enum dma_data_direction direction);
```

在这里,e size和direction参数必须匹配用于m AP缓冲区。

Some important rules apply to streaming DMA mappings:

- The buffer must be used only for a transfer that matches the direction value given when it was mapped.
- Once a buffer has been mapped, it belongs to the device, not the processor. Until the buffer has been unmapped, the driver should not touch its contents in any way. Only after *dma_unmap_single* has been called is it safe for the driver to access the contents of the buffer (with one exception that we see shortly). Among other things, this rule implies that a buffer being written to a device cannot be mapped until it contains all the data to write.
- The buffer must not be unmapped while DMA is still active, or serious system instability is guaranteed.

You may be wondering why the driver can no longer work with a buffer once it has been mapped. There are actually two reasons why this rule makes sense. First, when a buffer is mapped for DMA, the kernel must ensure that all of the data in that buffer has actually been written to memory. It is likely that some data is in the processor's cache when *dma_unmap_single* is issued, and must be explicitly flushed. Data written to the buffer by the processor after the flush may not be visible to the device.

Second, consider what happens if the buffer to be mapped is in a region of memory that is not accessible to the device. Some architectures simply fail in this case, but others create a bounce buffer. The bounce buffer is just a separate region of memory that *is* accessible to the device. If a buffer is mapped with a direction of `DMA_TO_DEVICE`, and a bounce buffer is required, the contents of the original buffer are copied as part of the mapping operation. Clearly, changes to the original buffer after the copy are not seen by the device. Similarly, `DMA_FROM_DEVICE` bounce buffers are copied back to the original buffer by *dma_unmap_single*; the data from the device is not present until that copy has been done.

Incidentally, bounce buffers are one reason why it is important to get the direction right. `DMA_BIDIRECTIONAL` bounce buffers are copied both before and after the operation, which is often an unnecessary waste of CPU cycles.

Occasionally a driver needs to access the contents of a streaming DMA buffer without unmapping it. A call has been provided to make this possible:

```
void dma_sync_single_for_cpu(struct device *dev, dma_handle_t bus_addr,
                             size_t size, enum dma_data_direction direction);
```

This function should be called before the processor accesses a streaming DMA buffer. Once the call has been made, the CPU "owns" the DMA buffer and can work with it as needed. Before the device accesses the buffer, however, ownership should be transferred back to it with:

```
void dma_sync_single_for_device(struct device *dev, dma_handle_t bus_addr,
                                size_t size, enum dma_data_direction direction);
```

一些重要规则适用于流媒体映射：

- 缓冲区必须仅用于与映射时给定的方向值匹配的传输。
- 映射缓冲区后，它属于设备，而不是处理器。直到缓冲区未盖上，驾驶员不应以任何方式触摸其内容。只有在*dma_unmap_single*被调用之后，驾驶员才能安全访问缓冲区的内容（我们很快就会看到一个例外）。除其他事项外，该规则暗示，在包含所有要编写的数据之前，将其写入设备的缓冲区可能不会被映射。
- 在DMA仍处于活动状态时，缓冲区不得将其解开，或者保证了严重的系统不稳定性。

您可能想知道，一旦映射，驾驶员为什么无法再用缓冲区工作。实际上，该规则有意义的原因有两个。首先，当为DMA映射缓冲区时，内核必须确保该缓冲区中的所有数据实际上已写入内存。当发出*dma_unmap_single*时，可能会在处理器的缓存中某些数据，并且必须明确刷新。在设备上可能看不到冲洗后的处理器对缓冲区的数据写入。

其次，考虑如果要映射的缓冲区在设备无法访问的内存区域中会发生什么。在这种情况下，有些体系结构只是失败了，而另一些则创建了弹跳缓冲区。弹跳缓冲区只是设备可访问的*is*的单独内存区域。如果映射一个缓冲区的方向`DMA_TO_DEVICE`，并且需要弹跳缓冲区，则原始缓冲区的内容作为映射操作的一部分进行了应对。显然，设备看不到副本后对原始缓冲区的更改。同样，`DMA_FROM_DEVICE`弹跳缓冲区被*dma_unmap_single*返回原始缓冲区;在完成该副本之前，设备的数据不存在。

顺便说一句，弹跳缓冲区是为什么正确的方向很重要的原因之一。`DMA_BIDIRECTIONAL`弹跳缓冲区在操作前后都被复制，这通常是不必要的CPU周期浪费。

有时，驱动程序需要访问流媒体DMA缓冲区的内容，并没有启用它。已经提供了通话以使其成为可能：

```
void dma_sync_single_for_cpu(struct device *dev, dma_handle_t bus_addr,
                             size_t size, enum dma_data_direction direction);
```

该功能应在处理器访问流DMA缓冲区之前调用。呼叫后，CPU将“拥有” DMA缓冲区，并可以根据需要使用它。但是，在设备访问缓冲区之前，应将所有权转移到它：

```
void dma_sync_single_for_device(struct device *dev, dma_handle_t bus_addr,
                                size_t size, enum dma_data_direction direction);
```

The processor, once again, should not access the DMA buffer after this call has been made.

Single-page streaming mappings

Occasionally, you may want to set up a mapping on a buffer for which you have a `struct page` pointer; this can happen, for example, with user-space buffers mapped with *get_user_pages*. To set up and tear down streaming mappings using `struct page` pointers, use the following:

```
dma_addr_t dma_map_page(struct device *dev, struct page *page,
                        unsigned long offset, size_t size,
                        enum dma_data_direction direction);

void dma_unmap_page(struct device *dev, dma_addr_t dma_address,
                    size_t size, enum dma_data_direction direction);
```

The `offset` and `size` arguments can be used to map part of a page. It is recommended, however, that partial-page mappings be avoided unless you are really sure of what you are doing. Mapping part of a page can lead to cache coherency problems if the allocation covers only part of a cache line; that, in turn, can lead to memory corruption and extremely difficult-to-debug bugs.

Scatter/gather mappings

Scatter/gather mappings are a special type of streaming DMA mapping. Suppose you have several buffers, all of which need to be transferred to or from the device. This situation can come about in several ways, including from a *readv* or *writev* system call, a clustered disk I/O request, or a list of pages in a mapped kernel I/O buffer. You could simply map each buffer, in turn, and perform the required operation, but there are advantages to mapping the whole list at once.

Many devices can accept a *scatterlist* of array pointers and lengths, and transfer them all in one DMA operation; for example, "zero-copy" networking is easier if packets can be built in multiple pieces. Another reason to map scatterlists as a whole is to take advantage of systems that have mapping registers in the bus hardware. On such systems, physically discontiguous pages can be assembled into a single, contiguous array from the device's point of view. This technique works only when the entries in the scatterlist are equal to the page size in length (except the first and last), but when it does work, it can turn multiple operations into a single DMA, and speed things up accordingly.

Finally, if a bounce buffer must be used, it makes sense to coalesce the entire list into a single buffer (since it is being copied anyway).

So now you're convinced that mapping of scatterlists is worthwhile in some situations. The first step in mapping a scatterlist is to create and fill in an array of `struct scatterlist` describing the buffers to be transferred. This structure is architecture

处理器再次不应访问此通话后不应访问DMA缓冲区。

单页流映射

有时，您可能需要在带有struct page指针的缓冲区上设置映射；例如，使用*get_user_pages*映射的用户空间缓冲区可能会发生这种情况。要使用struct page指针设置和拆除流映射，请使用以下内容：

```
dma_addr_t dma_map_page(struct device *dev, struct page *page,
                        unsigned long offset, size_t size,
                        enum dma_data_direction direction);

void dma_unmap_page(struct device *dev, dma_addr_t dma_address,
                    size_t size, enum dma_data_direction direction);
```

offset和size参数可用于映射页面的一部分。但是，建议您避免使用部分页面映射，除非您真的确定自己在做什么。如果分配仅覆盖缓存线的一部分，则页面的映射部分可能会导致缓存相干概率。反过来，这可能会导致犯罪腐败和极难挑剔的错误。

分散/收集映射

分散/收集映射是流媒体DMA映射的一种特殊类型。假设您有几个缓冲区，所有缓冲区都需要从设备转移或从设备转移。这种情况可以通过几种方式出现，包括*readv*或*writev*系统调用，群集磁盘I/O请求或映射的内核I/O Buffer中的页面列表。您可以依次简单地映射每个缓冲区，然后执行所需的操作，但是可以立即映射整个列表的优点。

许多设备可以接受数组指针和长度的*scatterlist*，并以一个DMA操作将它们全部传输；例如，如果可以用多个部分构建数据包，则“零拷贝”网络更容易。整体上绘制散落清单的另一个原因是利用在总线硬件中具有映射寄存器的系统。在这样的系统上，从设备的角度来看，可以将物理不连续的页面组装成一个连续的数组。此技术仅在散点表中的条目等于长度的页面大小（第一个和最后一个）时才起作用，但是当它确实有效时，它可以将多个操作变成一个单个DMA，并相应地加快速度。

最后，如果必须使用弹跳缓冲液，则将整个列表合并为一个缓冲区是有意义的（因为无论如何它正在复制）。

因此，现在您确信在某些情况下，散落清单的映射值得。映射散点表的第一步是创建并填充struct scatterlist的数组，描述要传输的缓冲区。这种结构是建筑

dependent, and is described in <*asm/scatterlist.h*>. However, it always contains three fields:

`struct page *page;`
: The `struct page` pointer corresponding to the buffer to be used in the scatter/gather operation.

`unsigned int length;`
`unsigned int offset;`
: The length of that buffer and its offset within the page

To map a scatter/gather DMA operation, your driver should set the `page`, `offset`, and `length` fields in a `struct scatterlist` entry for each buffer to be transferred. Then call:

```
int dma_map_sg(struct device *dev, struct scatterlist *sg, int nents,
               enum dma_data_direction direction)
```

where `nents` is the number of scatterlist entries passed in. The return value is the number of DMA buffers to transfer; it may be less than `nents`.

For each buffer in the input scatterlist, *dma_map_sg* determines the proper bus address to give to the device. As part of that task, it also coalesces buffers that are adjacent to each other in memory. If the system your driver is running on has an I/O memory management unit, *dma_map_sg* also programs that unit's mapping registers, with the possible result that, from your device's point of view, you are able to transfer a single, contiguous buffer. You will never know what the resulting transfer will look like, however, until after the call.

Your driver should transfer each buffer returned by *pci_map_sg*. The bus address and length of each buffer are stored in the `struct scatterlist` entries, but their location in the structure varies from one architecture to the next. Two macros have been defined to make it possible to write portable code:

`dma_addr_t sg_dma_address(struct scatterlist *sg);`
: Returns the bus (DMA) address from this scatterlist entry.

`unsigned int sg_dma_len(struct scatterlist *sg);`
: Returns the length of this buffer.

Again, remember that the address and length of the buffers to transfer may be different from what was passed in to *dma_map_sg*.

Once the transfer is complete, a scatter/gather mapping is unmapped with a call to *dma_unmap_sg*:

```
void dma_unmap_sg(struct device *dev, struct scatterlist *list,
                  int nents, enum dma_data_direction direction);
```

Note that `nents` must be the number of entries that you originally passed to *dma_map_sg* and not the number of DMA buffers the function returned to you.

依赖，并在<*asm/scatterlist.h*>中描述。但是，它总是包含三个字段：

`struct page *page;`
: struct page指针对应于用于散点/收集操作中的缓冲区。

`unsigned int length;`
`unsigned int offset;`
: 该缓冲区的长度及其在页面中的偏移

要映射散点/收集DMA操作，您的驱动程序应设置page，offset和length字段 struct scatterlist的字段struct scatterlist输入中的每个缓冲区。然后致电：

```
int dma_map_sg(struct device *dev, struct scatterlist *sg, int nents,
               enum dma_data_direction direction)
```

其中nents是传递的散点表条目的数量。返回值是要传输的DMA缓冲区的数量；它可能小于nents。

对于输入散点图中的每个缓冲区，*dma_map_sg*确定适当的总线地址以给设备。作为该任务的一部分，它还合并了在内存中相邻的缓冲区。如果您的驱动程序正在运行的系统具有I/O内存管理单元，则*dma_map_sg*还编程了单元映射重新配置的程序，从您的设备的角度来看，您可以转移一个连续的缓冲区。但是，直到通话后，您将永远不会知道所产生的转移会是什么样。

您的驱动程序应传输*pci_map_sg*返回的每个缓冲区。每个缓冲区的总线地址和长度存储在struct scatterlist条目中，但是它们在结构中的位置从一个体系结构到另一个体系结构变化。已经定义了两个宏，以便编写便携式代码：

`dma_addr_t sg_dma_address(struct scatterlist *sg);`
: 从此ScatterList条目返回公共汽车（DMA）地址 。

`unsigned int sg_dma_len(struct scatterlist *sg);`
: 返回此缓冲区的长度。

同样，请记住，要转移的缓冲区的地址和长度可能与传递给*dma_map_sg*的地址不同。

转移完成后，散布/收集映射将未上映，并呼叫*dma_unmap_sg*：

```
void dma_unmap_sg(struct device *dev, struct scatterlist *list,
                  int nents, enum dma_data_direction direction);
```

请注意，nents必须是您最初传递给*dma_map_sg*的条目数，而不是DMA缓冲区的数量，返回给您的函数。

Scatter/gather mappings are streaming DMA mappings, and the same access rules apply to them as to the single variety. If you must access a mapped scatter/gather list, you must synchronize it first:

```
void dma_sync_sg_for_cpu(struct device *dev, struct scatterlist *sg,
                         int nents, enum dma_data_direction direction);
void dma_sync_sg_for_device(struct device *dev, struct scatterlist *sg,
                         int nents, enum dma_data_direction direction);
```

PCI double-address cycle mappings

Normally, the DMA support layer works with 32-bit bus addresses, possibly restricted by a specific device's DMA mask. The PCI bus, however, also supports a 64-bit addressing mode, the *double-address cycle* (DAC). The generic DMA layer does not support this mode for a couple of reasons, the first of which being that it is a PCI-specific feature. Also, many implementations of DAC are buggy at best, and, because DAC is slower than a regular, 32-bit DMA, there can be a performance cost. Even so, there are applications where using DAC can be the right thing to do; if you have a device that is likely to be working with very large buffers placed in high memory, you may want to consider implementing DAC support. This support is available only for the PCI bus, so PCI-specific routines must be used.

To use DAC, your driver must include *<linux/pci.h>*. You must set a separate DMA mask:

```
int pci_dac_set_dma_mask(struct pci_dev *pdev, u64 mask);
```

You can use DAC addressing only if this call returns `0`.

A special type (`dma64_addr_t`) is used for DAC mappings. To establish one of these mappings, call *pci_dac_page_to_dma*:

```
dma64_addr_t pci_dac_page_to_dma(struct pci_dev *pdev, struct page *page,
                                 unsigned long offset, int direction);
```

DAC mappings, you will notice, can be made only from `struct page` pointers (they should live in high memory, after all, or there is no point in using them); they must be created a single page at a time. The `direction` argument is the PCI equivalent of the `enum dma_data_direction` used in the generic DMA layer; it should be `PCI_DMA_TODEVICE`, `PCI_DMA_FROMDEVICE`, or `PCI_DMA_BIDIRECTIONAL`.

DAC mappings require no external resources, so there is no need to explicitly release them after use. It is necessary, however, to treat DAC mappings like other streaming mappings, and observe the rules regarding buffer ownership. There is a set of functions for synchronizing DMA buffers that is analogous to the generic variety:

```
void pci_dac_dma_sync_single_for_cpu(struct pci_dev *pdev,
                                     dma64_addr_t dma_addr,
                                     size_t len,
                                     int direction);
```

散点/收集映射是流式DMA映射，并且适用于单个品种的访问规则。如果您必须访问映射的散点/收集列表，则必须先对其进行同步：

```
void dma_sync_sg_for_cpu(struct device *dev, struct scatterlist *sg,
                         int nents, enum dma_data_direction direction);
void dma_sync_sg_for_device(struct device *dev, struct scatterlist *sg,
                         int nents, enum dma_data_direction direction);
```

PCI双重地址循环映射

通常，DMA支持层可与32位总线地址一起使用，可能受到特定设备的DMA掩码的限制。但是，PCI总线还支持64位寻址模式，即*double-address cycle*（DAC）。通用DMA层不支持此模式的原因有两个原因，首先是它是PCI特定的功能。此外，许多DAC的实现充其量是越野车，并且由于DAC比常规的32位DMA慢，因此可能会有性能成本。即使这样，在某些应用程序中，使用DAC可能是正确的事情。如果您的设备可能正在使用放置在高内存中的非常大的缓冲区，则可能需要考虑实现DAC支持。此支持仅适用于PCI总线，因此必须使用PCI特定的例程。

要使用DAC，您的驱动程序必须包括<*linux/pci.h*>。您必须设置一个单独的DMA蒙版：

```
int pci_dac_set_dma_mask(struct pci_dev *pdev, u64 mask);
```

您只能在此调用返回0时才使用DAC地址。

特殊类型（dma64_addr_t）用于DAC映射。要建立这些映射之一，请致电*pci_dac_page_to_dma*：

```
dma64_addr_t pci_dac_page_to_dma(struct pci_dev *pdev, struct page *page,
                                 unsigned long offset, int direction);
```

您会注意到的DAC映射只能通过`struct page`指针进行（毕竟它们应该生活在高内存中，或者使用它们没有意义）；它们必须一次创建一个页面。`direction`参数是通用DMA层中使用的`enum dma_data_direction`的PCI等效；它应该是`PCI_DMA_TODEVICE`，`PCI_DMA_FROMDEVICE`或`PCI_DMA_BIDIRECTIONAL`。

DAC映射不需要外部资源，因此无需在使用后明确释放它们。但是，有必要像其他流映射一样对待DAC映射，并观察有关缓冲区所有权的规则。有一系列用于同步DMA缓冲区的功能，类似于通用品种：

```
void pci_dac_dma_sync_single_for_cpu(struct pci_dev *pdev,
                                     dma64_addr_t dma_addr,
                                     size_t len,
                                     int direction);
```

```
void pci_dac_dma_sync_single_for_device(struct pci_dev *pdev,
                                        dma64_addr_t dma_addr,
                                        size_t len,
                                        int direction);
```

A simple PCI DMA example

As an example of how the DMA mappings might be used, we present a simple example of DMA coding for a PCI device. The actual form of DMA operations on the PCI bus is very dependent on the device being driven. Thus, this example does not apply to any real device; instead, it is part of a hypothetical driver called *dad* (DMA Acquisition Device). A driver for this device might define a transfer function like this:

```
int dad_transfer(struct dad_dev *dev, int write, void *buffer,
                 size_t count)
{
    dma_addr_t bus_addr;

    /* Map the buffer for DMA */
    dev->dma_dir = (write ? DMA_TO_DEVICE : DMA_FROM_DEVICE);
    dev->dma_size = count;
    bus_addr = dma_map_single(&dev->pci_dev->dev, buffer, count,
                              dev->dma_dir);
    dev->dma_addr = bus_addr;

    /* Set up the device */

    writeb(dev->registers.command, DAD_CMD_DISABLEDMA);
    writeb(dev->registers.command, write ? DAD_CMD_WR : DAD_CMD_RD);
    writel(dev->registers.addr, cpu_to_le32(bus_addr));
    writel(dev->registers.len, cpu_to_le32(count));

    /* Start the operation */
    writeb(dev->registers.command, DAD_CMD_ENABLEDMA);
    return 0;
}
```

This function maps the buffer to be transferred and starts the device operation. The other half of the job must be done in the interrupt service routine, which looks something like this:

```
void dad_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
    struct dad_dev *dev = (struct dad_dev *) dev_id;

    /* Make sure it's really our device interrupting */

    /* Unmap the DMA buffer */
    dma_unmap_single(dev->pci_dev->dev, dev->dma_addr,
                     dev->dma_size, dev->dma_dir);

    /* Only now is it safe to access the buffer, copy to user, etc. */
    ...
}
```

```
void pci_dac_dma_sync_single_for_device(struct pci_dev *pdev,
                                        dma64_addr_t dma_addr,
                                        size_t len,
                                        int direction);
```

一个简单的PCI DMA示例

作为如何使用DMA映射的一个示例，我们为PCI设备提供了简单的DMA编码检查。PCI总线上DMA操作的实际形式非常取决于所驱动的设备。因此，此示例不适用于任何真实设备。相反，它是一个假设驱动程序的一部分，称为*dad*（DMA获取设备)。该设备的驱动程序可能会定义这样的传输函数:

```
int dad_transfer(struct dad_dev *dev, int write, void *buffer,
                 size_t count)
{
    dma_addr_t bus_addr;

    /* Map the buffer for DMA */
    dev->dma_dir = (write ? DMA_TO_DEVICE : DMA_FROM_DEVICE);
    dev->dma_size = count;
    bus_addr = dma_map_single(&dev->pci_dev->dev, buffer, count,
                              dev->dma_dir);
    dev->dma_addr = bus_addr;

    /* Set up the device */

    writeb(dev->registers.command, DAD_CMD_DISABLEDMA);
    writeb(dev->registers.command, write ? DAD_CMD_WR : DAD_CMD_RD);
    writel(dev->registers.addr, cpu_to_le32(bus_addr));
    writel(dev->registers.len, cpu_to_le32(count));

    /* Start the operation */
    writeb(dev->registers.command, DAD_CMD_ENABLEDMA);
    return 0;
}
```

此功能映射要传输的缓冲区并启动设备操作。工作的另一半必须在中断服务程序中完成，这看起来像这样:

```
void dad_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
    struct dad_dev *dev = (struct dad_dev *) dev_id;

    /* Make sure it's really our device interrupting */

    /* Unmap the DMA buffer */
    dma_unmap_single(dev->pci_dev->dev, dev->dma_addr,
                     dev->dma_size, dev->dma_dir);

    /* Only now is it safe to access the buffer, copy to user, etc. */
    ...
}
```

Obviously, a great deal of detail has been left out of this example, including whatever steps may be required to prevent attempts to start multiple, simultaneous DMA operations.

DMA for ISA Devices

The ISA bus allows for two kinds of DMA transfers: native DMA and ISA bus master DMA. Native DMA uses standard DMA-controller circuitry on the motherboard to drive the signal lines on the ISA bus. ISA bus master DMA, on the other hand, is handled entirely by the peripheral device. The latter type of DMA is rarely used and doesn't require discussion here, because it is similar to DMA for PCI devices, at least from the driver's point of view. An example of an ISA bus master is the 1542 SCSI controller, whose driver is *drivers/scsi/aha1542.c* in the kernel sources.

As far as native DMA is concerned, there are three entities involved in a DMA data transfer on the ISA bus:

The 8237 DMA controller (DMAC)
: The controller holds information about the DMA transfer, such as the direction, the memory address, and the size of the transfer. It also contains a counter that tracks the status of ongoing transfers. When the controller receives a DMA request signal, it gains control of the bus and drives the signal lines so that the device can read or write its data.

The peripheral device
: The device must activate the DMA request signal when it's ready to transfer data. The actual transfer is managed by the DMAC; the hardware device sequentially reads or writes data onto the bus when the controller strobes the device. The device usually raises an interrupt when the transfer is over.

The device driver
: The driver has little to do; it provides the DMA controller with the direction, bus address, and size of the transfer. It also talks to its peripheral to prepare it for transferring the data and responds to the interrupt when the DMA is over.

The original DMA controller used in the PC could manage four "channels," each associated with one set of DMA registers. Four devices could store their DMA information in the controller at the same time. Newer PCs contain the equivalent of two DMAC devices:* the second controller (master) is connected to the system processor, and the first (slave) is connected to channel `0` of the second controller.†

* These circuits are now part of the motherboard's chipset, but a few years ago they were two separate 8237 chips.

† The original PCs had only one controller; the second was added in 286-based platforms. However, the second controller is connected as the master because it handles 16-bit transfers; the first transfers only eight bits at a time and is there for backward compatibility.

显然，在此示例中遗漏了大量细节，包括可能需要采取任何措施来防止尝试同时进行多次DMA操作。

ISA设备的DMA

ISA总线允许两种DMA转移：天然DMA和ISA总线Master DMA。天然DMA在主板上使用标准的DMA控制器电路来驱动ISA总线上的信号线。另一方面，ISA总线大师DMA完全由外围设备处理。后一种DMA很少使用，并且在这里不需要讨论，因为它与PCI设备的DMA相似，至少从驾驶员的角度来看。 ISA总线大师的一个示例是1542 SCSI控制器，其驱动程序是内核源中的*drivers/scsi/aha1542.c*。

就本地DMA而言，ISA总线上的DMA数据传输中涉及三个实体：

The 8237 DMA controller (DMAC)
: 控制器保留了有关DMA传输的信息，例如方向，内存地址和传输的大小。它还包含一个跟踪正在进行的转移状态的计数器。当控制器收到DMA请求信号时，它会获得对总线的控制并驱动信号线，以便设备可以读取或写入其数据。

The peripheral device
: 设备准备传输数据时必须激活DMA请求信号。实际转移由DMAC管理；当控制器将设备带到设备时，硬件设备的序列序列将数据读取或将数据写入总线上。传输结束时，该设备通常会引起中断。

The device driver
: 驾驶员几乎没有什么事。它为DMA控制器提供了转移的方向，总线地址和大小。它还与它的外围交谈以准备传输数据并在DMA结束时响应中断。

PC中使用的原始DMA控制器可以管理四个“通道”，每组与一组DMA寄存器相关联。四个设备可以同时将其DMA信息存储在控制器中。较新的PC包含两个DMAC设备的等效：*第二控制器（Master）连接到系统proces-sor，并且第一个（从）连接到第二控制器的通道0。†。

* These circuits are now part of the motherboard’s chipset, but a few years ago they were two separate 8237 chips.

† The original PCs had only one controller; the second was added in 286-based platforms. However, the second controller is connected as the master because it handles 16-bit transfers; the first transfers only eight bits at a time and is there for backward compatibility.

The channels are numbered from 0–7: channel 4 is not available to ISA peripherals, because it is used internally to cascade the slave controller onto the master. The available channels are, thus, 0–3 on the slave (the 8-bit channels) and 5–7 on the master (the 16-bit channels). The size of any DMA transfer, as stored in the controller, is a 16-bit number representing the number of bus cycles. The maximum transfer size is, therefore, 64 KB for the slave controller (because it transfers eight bits in one cycle) and 128 KB for the master (which does 16-bit transfers).

Because the DMA controller is a system-wide resource, the kernel helps deal with it. It uses a DMA registry to provide a request-and-free mechanism for the DMA channels and a set of functions to configure channel information in the DMA controller.

Registering DMA usage

You should be used to kernel registries—we've already seen them for I/O ports and interrupt lines. The DMA channel registry is similar to the others. After *<asm/dma.h>* has been included, the following functions can be used to obtain and release ownership of a DMA channel:

```
int request_dma(unsigned int channel, const char *name);
void free_dma(unsigned int channel);
```

The `channel` argument is a number between 0 and 7 or, more precisely, a positive number less than `MAX_DMA_CHANNELS`. On the PC, `MAX_DMA_CHANNELS` is defined as `8` to match the hardware. The `name` argument is a string identifying the device. The specified name appears in the file */proc/dma*, which can be read by user programs.

The return value from *request_dma* is `0` for success and `-EINVAL` or `-EBUSY` if there was an error. The former means that the requested channel is out of range, and the latter means that another device is holding the channel.

We recommend that you take the same care with DMA channels as with I/O ports and interrupt lines; requesting the channel at *open* time is much better than requesting it from the module initialization function. Delaying the request allows some sharing between drivers; for example, your sound card and your analog I/O interface can share the DMA channel as long as they are not used at the same time.

We also suggest that you request the DMA channel *after* you've requested the interrupt line and that you release it *before* the interrupt. This is the conventional order for requesting the two resources; following the convention avoids possible deadlocks. Note that every device using DMA needs an IRQ line as well; otherwise, it couldn't signal the completion of data transfer.

In a typical case, the code for *open* looks like the following, which refers to our hypothetical *dad* module. The *dad* device as shown uses a fast interrupt handler without support for shared IRQ lines.

```
int dad_open (struct inode *inode, struct file *filp)
{
    struct dad_device *my_device;
```

频道从0-7：频道4编号为ISA外围设备，因为它在内部用于将从属控制器级联到主机上。因此，可用的通道在从（8位通道）上为0-3，主（16位通道）上的通道为5-7。任何DMA转移的大小（存储在对照组中）是一个16位数字，代表总线循环的数量。因此，对从控制器的最大跨大小为64 kb（因为它在一个周期内传输八位）和128 kb的主人（可进行16位传输）。

由于DMA控制器是全系统资源，因此内核有助于处理它。它使用DMA注册表为DMA Channels提供无请求和无请求机制，并在DMA控制器中配置通道信息。

注册DMA使用情况

您应该习惯于内核注册表，我们已经看到了它们的I/O端口和中断线路。 DMA渠道注册表与其他渠道相似。包括<*asm/dma.h*>后，可以使用以下功能来获取和发布DMA渠道的所有者：

```
int request_dma(unsigned int channel, const char *name);
void free_dma(unsigned int channel);
```

`channel`参数是0到7之间的数字，或者更确切地说，一个比`MAX_DMA_CHANNELS`小的正数。在PC上，`MAX_DMA_CHANNELS`定义为8以匹配硬件。 `name`参数是标识设备的字符串。指定名称出现在文件*/proc/dma*中，可以通过用户程序读取。

如果有错误，则来自*request_dma*的返回值是`0`的`0`，`-EINVAL`或`-EBUSY`是错误的。前者意味着请求的通道超出范围，而后者则意味着另一个设备持有该通道。

我们建议您使用DMA频道与I/O端口和中断线相同的护理；在*open*时间请求通道比从模块初始化函数请求要好得多。延迟该请求可以在驱动程序之间进行一些分享；例如，只要不同时使用，您的声卡和模拟I/O接口就可以共享DMA频道。

我们还建议您请求DMA频道*after*您已请求间断行，并将其发布*before*中断。这是要求这两个资源的常规订单；遵循大会避免可能的死锁。请注意，使用DMA的每个设备也需要IRQ线路；否则，它无法表示数据传输的完成。

在典型的情况下，*open*的代码看起来如下，它指的是我们的弱点*dad*模块。如图所示的*dad*设备使用一个快速中断处理程序，而无需支持共享IRQ线路。

```
int dad_open (struct inode *inode, struct file *filp)
{
    struct dad_device *my_device;
```

```
    /* ... */
    if ( (error = request_irq(my_device.irq, dad_interrupt,
                              SA_INTERRUPT, "dad", NULL)) )
        return error; /* or implement blocking open */

    if ( (error = request_dma(my_device.dma, "dad")) ) {
        free_irq(my_device.irq, NULL);
        return error; /* or implement blocking open */
    }
    /* ... */
    return 0;
}
```

The *close* implementation that matches the *open* just shown looks like this:

```
void dad_close (struct inode *inode, struct file *filp)
{
    struct dad_device *my_device;

    /* ... */
    free_dma(my_device.dma);
    free_irq(my_device.irq, NULL);
    /* ... */
}
```

Here's how the */proc/dma* file looks on a system with the sound card installed:

```
merlino% cat /proc/dma
 1: Sound Blaster8
 4: cascade
```

It's interesting to note that the default sound driver gets the DMA channel at system boot and never releases it. The `cascade` entry is a placeholder, indicating that channel 4 is not available to drivers, as explained earlier.

Talking to the DMA controller

After registration, the main part of the driver's job consists of configuring the DMA controller for proper operation. This task is not trivial, but fortunately, the kernel exports all the functions needed by the typical driver.

The driver needs to configure the DMA controller either when *read* or *write* is called, or when preparing for asynchronous transfers. This latter task is performed either at *open* time or in response to an *ioctl* command, depending on the driver and the policy it implements. The code shown here is the code that is typically called by the *read* or *write* device methods.

This subsection provides a quick overview of the internals of the DMA controller so you understand the code introduced here. If you want to learn more, we'd urge you to read <*asm/dma.h*> and some hardware manuals describing the PC architecture. In

```
        /* ... */
        if ( (error = request_irq(my_device.irq, dad_interrupt,
                                  SA_INTERRUPT, "dad", NULL)) )
            return error; /* or implement blocking open */

        if ( (error = request_dma(my_device.dma, "dad")) ) {
            free_irq(my_device.irq, NULL);
            return error; /* or implement blocking open */
        }
        /* ... */
        return 0;
    }
```

与*open*匹配的*close*实现恰好如下所示：

```
void dad_close (struct inode *inode, struct file *filp)
{
    struct dad_device *my_device;

    /* ... */
    free_dma(my_device.dma);
    free_irq(my_device.irq, NULL);
    /* ... */
}
```

以下是*/proc/dma*文件在系统上使用声卡安装的系统的方式：

```
merlino% cat /proc/dma
 1: Sound Blaster8
 4: cascade
```

有趣的是，默认的声音驱动程序在系统启动处获取DMA通道，并且永远不会发布。cascade条目是占位符，表明司机无法使用Channel 4，如前所述。

与DMA控制器交谈

注册后，驾驶员作业的主要部分包括配置DMA控制器以进行正常操作。此任务并不小，但幸运的是，内核导出了典型驱动程序所需的所有功能。

当调用*read*或*write*时，或者在准备异步传输时，驱动程序需要配置DMA控制器。后一个任务是在*open*时间或响应*ioctl*命令时执行的，具体取决于驱动程序和policy IT的实现。此处显示的代码是通常由*read*或*write*设备方法调用的代码。

该小节可快速概述DMA控制器的内部词，因此您了解此处介绍的代码。如果您想了解更多信息，我们敦促您阅读*<asm/dma.h>*和一些描述PC体系结构的硬件手册。在

particular, we don't deal with the issue of 8-bit versus 16-bit data transfers. If you are writing device drivers for ISA device boards, you should find the relevant information in the hardware manuals for the devices.

The DMA controller is a shared resource, and confusion could arise if more than one processor attempts to program it simultaneously. For that reason, the controller is protected by a spinlock, called `dma_spin_lock`. Drivers should not manipulate the lock directly; however, two functions have been provided to do that for you:

`unsigned long claim_dma_lock( );`
: Acquires the DMA spinlock. This function also blocks interrupts on the local processor; therefore, the return value is a set of flags describing the previous interrupt state; it must be passed to the following function to restore the interrupt state when you are done with the lock.

`void release_dma_lock(unsigned long flags);`
: Returns the DMA spinlock and restores the previous interrupt status.

The spinlock should be held when using the functions described next. It should *not* be held during the actual I/O, however. A driver should never sleep when holding a spinlock.

The information that must be loaded into the controller consists of three items: the RAM address, the number of atomic items that must be transferred (in bytes or words), and the direction of the transfer. To this end, the following functions are exported by *<asm/dma.h>*:

`void set_dma_mode(unsigned int channel, char mode);`
: Indicates whether the channel must read from the device (`DMA_MODE_READ`) or write to it (`DMA_MODE_WRITE`). A third mode exists, `DMA_MODE_CASCADE`, which is used to release control of the bus. Cascading is the way the first controller is connected to the top of the second, but it can also be used by true ISA bus-master devices. We won't discuss bus mastering here.

`void set_dma_addr(unsigned int channel, unsigned int addr);`
: Assigns the address of the DMA buffer. The function stores the 24 least significant bits of `addr` in the controller. The `addr` argument must be a *bus* address (see the section "Bus Addresses" earlier in this chapter).

`void set_dma_count(unsigned int channel, unsigned int count);`
: Assigns the number of bytes to transfer. The `count` argument represents bytes for 16-bit channels as well; in this case, the number *must* be even.

特别是，我们不处理8位与16位数据传输的问题。如果您正在为ISA设备板编写设备驱动程序，则应在设备的硬件手册中找到相关信息。

DMA控制器是共享资源，如果多个处理器尝试同时编程，可能会引起混乱。因此，控制器由称为`dma_spin_lock`的旋转锁保护。驾驶员不应直接操纵锁；但是，已经提供了两个功能来为您做到这一点：

`unsigned long claim_dma_lock( );`
: 获取DMA Spinlock。此功能还阻止了本地处理器上的中断。因此，返回值是描述先前中断状态的一组标志。必须将其传递给以下功能，以恢复使用锁时的间断状态。

`void release_dma_lock(unsigned long flags);`
: 返回DMA Spinlock并恢复先前的中断状态。

使用下一个描述的函数时，应保持旋转锁。但是，应该在实际的I/O期间保留*not*。拿着自旋锁时，驾驶员永远不会睡觉。

必须加载到控制器的信息由三个项目组成：RAM地址，必须转移的原子项目数（字节或单词）以及转移的方向。为此，以下功能由<*asm/dma.h*>导出：

`void set_dma_mode(unsigned int channel, char mode);`
: 指示该通道是否必须从设备（`DMA_MODE_READ`）读取或写入（`DMA_MODE_WRITE`）。存在第三个模式，`DMA_MODE_CASCADE`，用于释放总线的控制。级联是第一个控制器连接到第二个控制器的方式，但也可以由True ISA Bus-Master设备使用。我们不会在这里讨论公共汽车的掌握。

`void set_dma_addr(unsigned int channel, unsigned int addr);`
: 分配DMA缓冲区的地址。该函数存储在控制器中`addr`的24位最小值位。 `addr`参数必须是*bus*地址（请参阅本章早期的“总线地址”部分）。

`void set_dma_count(unsigned int channel, unsigned int count);`
: 分配传输的字节数。 `count`参数也代表16位通道的字节；在这种情况下，数字*must*是偶数。

In addition to these functions, there are a number of housekeeping facilities that must be used when dealing with DMA devices:

`void disable_dma(unsigned int channel);`
: A DMA channel can be disabled within the controller. The channel should be disabled before the controller is configured to prevent improper operation. (Otherwise, corruption can occur because the controller is programmed via 8-bit data transfers and, therefore, none of the previous functions is executed atomically).

`void enable_dma(unsigned int channel);`
: This function tells the controller that the DMA channel contains valid data.

`int get_dma_residue(unsigned int channel);`
: The driver sometimes needs to know whether a DMA transfer has been completed. This function returns the number of bytes that are still to be transferred. The return value is `0` after a successful transfer and is unpredictable (but not `0`) while the controller is working. The unpredictability springs from the need to obtain the 16-bit residue through two 8-bit input operations.

`void clear_dma_ff(unsigned int channel)`
: This function clears the DMA flip-flop. The flip-flop is used to control access to 16-bit registers. The registers are accessed by two consecutive 8-bit operations, and the flip-flop is used to select the least significant byte (when it is clear) or the most significant byte (when it is set). The flip-flop automatically toggles when eight bits have been transferred; the programmer must clear the flip-flop (to set it to a known state) before accessing the DMA registers.

Using these functions, a driver can implement a function like the following to prepare for a DMA transfer:

```
int dad_dma_prepare(int channel, int mode, unsigned int buf,
                    unsigned int count)
{
    unsigned long flags;

    flags = claim_dma_lock( );
    disable_dma(channel);
    clear_dma_ff(channel);
    set_dma_mode(channel, mode);
    set_dma_addr(channel, virt_to_bus(buf));
    set_dma_count(channel, count);
    enable_dma(channel);
    release_dma_lock(flags);

    return 0;
}
```

Then, a function like the next one is used to check for successful completion of DMA:

```
int dad_dma_isdone(int channel)
{
```

除这些功能外，在处理DMA设备时，必须使用许多管家设施：

`void disable_dma(unsigned int channel);`
DMA通道可以在控制器中禁用。在配置控制器以防止操作不当之前，应禁用通道。（otherwise，可能会发生损坏，因为控制器是通过8位数据传输对控制器进行编程的，因此，先前的函数均未在原子上执行）。

`void enable_dma(unsigned int channel);`
这个功能 离子告诉控制器DMA通道包含有效 数据

。`int get_dma_residue(unsigned int channel);`
驾驶员有时需要知道是否已完成DMA转移。此功能返回仍将传输的字节数。成功传输后的返回值为0，并且在控制器正常工作时不可预测（但0）。不需要通过两个8位输入操作获得16位残留物的不可预测性。

`void clear_dma_ff(unsigned int channel)`
此函数清除了DMA触发器。触发器用于控制对16位寄存器的访问。寄存器由两个连续的8位操作访问，并使用触发器来选择最低的字节（当清晰时）或最重要的字节（设置时）。转移八位时，触发器会自动切换；程序员必须在访问DMA寄存器之前清除触发器（将其设置为已知状态）。

使用这些功能，驱动程序可以实现以下功能以预先进行DMA传输：

```
int dad_dma_prepare(int channel, int mode, unsigned int buf,
                    unsigned int count)
{
    unsigned long flags;

    flags = claim_dma_lock( );
    disable_dma(channel);
    clear_dma_ff(channel);
    set_dma_mode(channel, mode);
    set_dma_addr(channel, virt_to_bus(buf));
    set_dma_count(channel, count);
    enable_dma(channel);
    release_dma_lock(flags);

    return 0;
}
```

然后，使用下一个功能来检查DMA的成功完成：

```
int dad_dma_isdone(int channel)
{
```

```
        int residue;
        unsigned long flags = claim_dma_lock ();
        residue = get_dma_residue(channel);
        release_dma_lock(flags);
        return (residue == 0);
    }
```

The only thing that remains to be done is to configure the device board. This device-specific task usually consists of reading or writing a few I/O ports. Devices differ in significant ways. For example, some devices expect the programmer to tell the hardware how big the DMA buffer is, and sometimes the driver has to read a value that is hardwired into the device. For configuring the board, the hardware manual is your only friend.

Quick Reference

This chapter introduced the following symbols related to memory handling.

Introductory Material

#include <linux/mm.h>
#include <asm/page.h>
: Most of the functions and structures related to memory management are prototyped and defined in these header files.

void *__va(unsigned long physaddr);
unsigned long __pa(void *kaddr);
: Macros that convert between kernel logical addresses and physical addresses.

PAGE_SIZE
PAGE_SHIFT
: Constants that give the size (in bytes) of a page on the underlying hardware and the number of bits that a page frame number must be shifted to turn it into a physical address.

struct page
: Structure that represents a hardware page in the system memory map.

struct page *virt_to_page(void *kaddr);
void *page_address(struct page *page);
struct page *pfn_to_page(int pfn);
: Macros that convert between kernel logical addresses and their associated memory map entries. *page_address* works only for low-memory pages or high-memory pages that have been explicitly mapped. *pfn_to_page* converts a page frame number to its associated struct page pointer.

```
        int residue;
        unsigned long flags = claim_dma_lock ();
        residue = get_dma_residue(channel);
        release_dma_lock(flags);
        return (residue == 0);
    }
```

唯一要做的是配置设备板。此设备特定的任务通常包括读取或编写一些I/O端口。设备在很大程度上有所不同。例如，某些设备希望程序员告诉硬件DMA缓冲区有多大，有时驱动程序必须读取一个硬连接到设备中的值。为了配置板，硬件手册是您唯一的朋友。

快速参考

本章介绍了与内存处理有关的以下符号。

入门材料

`#include <linux/mm.h>`
`#include <asm/page.h>`

与内存管理相关的大多数功能和结构都是在这些标头文件中进行原始打字和定义的。 `void *__va(unsigned long physaddr);` `unsigned long __pa(void *kaddr);` 宏位于内核逻辑地址和物理地址之间。 `PAGE_SIZE` 常数在基础硬件上给出页面的大小（以字节为单位）， 以及必须将页面框架编号移动以将其转换为物理地址的位数。 `PAGE_SHIFT`

`struct page`

在系统内存映射中代表硬件页面的结构。

`struct page *virt_to_page(void *kaddr);`
`void *page_address(struct page *page);`
`struct page *pfn_to_page(int pfn);`

宏位于内核逻辑地址及其关联的内存地图条目之间。 *page_address*仅适用于已明确映射的低内存页面或高内存页面。 *pfn_to_page*将页面数字转换为其关联的`struct page`指针。

`unsigned long kmap(struct page *page);`
`void kunmap(struct page *page);`
: *kmap* returns a kernel virtual address that is mapped to the given page, creating the mapping if need be. *kunmap* deletes the mapping for the given page.

`#include <linux/highmem.h>`
`#include <asm/kmap_types.h>`
`void *kmap_atomic(struct page *page, enum km_type type);`
`void kunmap_atomic(void *addr, enum km_type type);`
: The high-performance version of *kmap*; the resulting mappings can be held only by atomic code. For drivers, `type` should be `KM_USER0`, `KM_USER1`, `KM_IRQ0`, or `KM_IRQ1`.

`struct vm_area_struct;`
: Structure describing a VMA.

Implementing mmap

`int remap_pfn_range(struct vm_area_struct *vma, unsigned long virt_add, unsigned long pfn, unsigned long size, pgprot_t prot);`
`int io_remap_page_range(struct vm_area_struct *vma, unsigned long virt_add, unsigned long phys_add, unsigned long size, pgprot_t prot);`
: Functions that sit at the heart of *mmap*. They map `size` bytes of physical addresses, starting at the page number indicated by `pfn` to the virtual address `virt_add`. The protection bits associated with the virtual space are specified in `prot`. *io_remap_page_range* should be used when the target address is in I/O memory space.

`struct page *vmalloc_to_page(void *vmaddr);`
: Converts a kernel virtual address obtained from *vmalloc* to its corresponding `struct page` pointer.

Implementing Direct I/O

`int get_user_pages(struct task_struct *tsk, struct mm_struct *mm, unsigned long start, int len, int write, int force, struct page **pages, struct vm_area_struct **vmas);`
: Function that locks a user-space buffer into memory and returns the corresponding `struct page` pointers. The caller must hold `mm->mmap_sem`.

`SetPageDirty(struct page *page);`
: Macro that marks the given page as “dirty” (modified) and in need of writing to its backing store before it can be freed.

`void page_cache_release(struct page *page);`
: Frees the given page from the page cache.

```
unsigned long kmap(struct page *page);
void kunmap(struct page *page);
```

kmap 返回映射到给定页面的内核虚拟地址，如果需要，可以创建映射。*kunmap*删除给定页面的映射。

```
#include <linux/highmem.h>
#include <asm/kmap_types.h>
void *kmap_atomic(struct page *page, enum km_type type);
void kunmap_atomic(void *addr, enum km_type type);
```

*kmap*的高性能版本;最终的映射只能通过原子代码进行。对于驱动程序，type应为KM_USER0, KM_USER1, KM_IRQ0或KM_IRQ1。 struct vm_area_struct;描述VMA的结构。

实施MMAP

```
int remap_pfn_range(struct vm_area_struct *vma, unsigned long virt_add,
  unsigned long pfn, unsigned long size, pgprot_t prot);
int io_remap_page_range(struct vm_area_struct *vma, unsigned long virt_add,
  unsigned long phys_add, unsigned long size, pgprot_t prot);
```

位于*mmap*核心的功能。他们将物理地址的size字节映射为pfn指示的页码开始到虚拟地址virt_add。 prot中指定了与虚拟空间相关的保护位。当目标地址在I/O内存空间中时，应使用*io_remap_page_range*。

```
struct page *vmalloc_to_page(void *vmaddr);
```

将从*vmalloc*获得的内核虚拟地址转换为相应的struct page指针。

实施直接I/O。

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

将用户空间缓冲区锁定到内存中并返回相应struct page指针的功能。呼叫者必须保持mm->mmap_sem。

```
SetPageDirty(struct page *page);
```

宏将给定页面标记为“脏”（修改），并且需要在释放其备份之前写入其支持商店。

```
void page_cache_release(struct page *page);
```

从页面缓存中释放给定页面。

`int is_sync_kiocb(struct kiocb *iocb);`
: Macro that returns nonzero if the given IOCB requires synchronous execution.

`int aio_complete(struct kiocb *iocb, long res, long res2);`
: Function that indicates completion of an asynchronous I/O operation.

Direct Memory Access

```
#include <asm/io.h>
unsigned long virt_to_bus(volatile void * address);
void * bus_to_virt(unsigned long address);
```
Obsolete and deprecated functions that convert between kernel, virtual, and bus addresses. Bus addresses must be used to talk to peripheral devices.

```
#include <linux/dma-mapping.h>
```
Header file required to define the generic DMA functions.

```
int dma_set_mask(struct device *dev, u64 mask);
```
For peripherals that cannot address the full 32-bit range, this function informs the kernel of the addressable range and returns nonzero if DMA is possible.

```
void *dma_alloc_coherent(struct device *dev, size_t size, dma_addr_t
  *bus_addr, int flag)
void dma_free_coherent(struct device *dev, size_t size, void *cpuaddr,
  dma_handle_t bus_addr);
```
Allocate and free coherent DMA mappings for a buffer that will last the lifetime of the driver.

```
#include <linux/dmapool.h>
struct dma_pool *dma_pool_create(const char *name, struct device *dev,
  size_t size, size_t align, size_t allocation);
void dma_pool_destroy(struct dma_pool *pool);
void *dma_pool_alloc(struct dma_pool *pool, int mem_flags, dma_addr_t
   *handle);
void dma_pool_free(struct dma_pool *pool, void *vaddr, dma_addr_t handle);
```
Functions that create, destroy, and use DMA pools to manage small DMA areas.

```
enum dma_data_direction;
DMA_TO_DEVICE
DMA_FROM_DEVICE
DMA_BIDIRECTIONAL
DMA_NONE
```
Symbols used to tell the streaming mapping functions the direction in which data is moving to or from the buffer.

```
int is_sync_kiocb(struct kiocb *iocb);
```

如果给定的IOCB需要同步执行，则返回非零的宏。

`int aio_complete(struct kiocb *iocb, long res, long res2);`函数，指示完成异步I/O操作的完成。

直接内存访问

```
#include <asm/io.h>
unsigned long virt_to_bus(volatile void * address);
void * bus_to_virt(unsigned long address);
```

在内核，虚拟和总线地址之间转换的过时和弃用功能。公交地址必须用于与外围设备交谈。

```
#include <linux/dma-mapping.h>
```

定义通用DMA功能所需的标头文件。

`int dma_set_mask(struct device *dev, u64 mask);`对于无法解决整个32位范围的外围设备，此功能会为内核提供可寻址范围，并在可能的情况下返回非零。

```
void *dma_alloc_coherent(struct device *dev, size_t size, dma_addr_t
  *bus_addr, int flag)
void dma_free_coherent(struct device *dev, size_t size, void *cpuaddr,
  dma_handle_t bus_addr);
```

为缓冲区分配和免费相干DMA映射，该缓冲区将持续驱动程序的使用寿命。

```
#include <linux/dmapool.h>
struct dma_pool *dma_pool_create(const char *name, struct device *dev,
  size_t size, size_t align, size_t allocation);
void dma_pool_destroy(struct dma_pool *pool);
void *dma_pool_alloc(struct dma_pool *pool, int mem_flags, dma_addr_t
  *handle);
void dma_pool_free(struct dma_pool *pool, void *vaddr, dma_addr_t handle);
```

函数创建，销毁和使用DMA池来管理小型DMA区域。

```
enum dma_data_direction;
DMA_TO_DEVICE
DMA_FROM_DEVICE
DMA_BIDIRECTIONAL
DMA_NONE
```

符号用于告诉流映射功能数据向缓冲区移动或从缓冲区移动的方向。

`dma_addr_t dma_map_single(struct device *dev, void *buffer, size_t size, enum dma_data_direction direction);`
`void dma_unmap_single(struct device *dev, dma_addr_t bus_addr, size_t size, enum dma_data_direction direction);`
: Create and destroy a single-use, streaming DMA mapping.

`void dma_sync_single_for_cpu(struct device *dev, dma_handle_t bus_addr, size_t size, enum dma_data_direction direction);`
`void dma_sync_single_for_device(struct device *dev, dma_handle_t bus_addr, size_t size, enum dma_data_direction direction);`
: Synchronizes a buffer that has a streaming mapping. These functions must be used if the processor must access a buffer while the streaming mapping is in place (i.e., while the device owns the buffer).

`#include <asm/scatterlist.h>`
`struct scatterlist { /* ... */ };`
`dma_addr_t sg_dma_address(struct scatterlist *sg);`
`unsigned int sg_dma_len(struct scatterlist *sg);`
: The `scatterlist` structure describes an I/O operation that involves more than one buffer. The macros *sg_dma_address* and *sg_dma_len* may be used to extract bus addresses and buffer lengths to pass to the device when implementing scatter/gather operations.

`dma_map_sg(struct device *dev, struct scatterlist *list, int nents, enum dma_data_direction direction);`
`dma_unmap_sg(struct device *dev, struct scatterlist *list, int nents, enum dma_data_direction direction);`
`void dma_sync_sg_for_cpu(struct device *dev, struct scatterlist *sg, int nents, enum dma_data_direction direction);`
`void dma_sync_sg_for_device(struct device *dev, struct scatterlist *sg, int nents, enum dma_data_direction direction);`
: *dma_map_sg* maps a scatter/gather operation, and *dma_unmap_sg* undoes that mapping. If the buffers must be accessed while the mapping is active, *dma_sync_sg_** may be used to synchronize things.

`/proc/dma`
: File that contains a textual snapshot of the allocated channels in the DMA controllers. PCI-based DMA is not shown because each board works independently, without the need to allocate a channel in the DMA controller.

`#include <asm/dma.h>`
: Header that defines or prototypes all the functions and macros related to DMA. It must be included to use any of the following symbols.

```
dma_addr_t dma_map_single(struct device *dev, void *buffer, size_t size, enum
  dma_data_direction direction);
void dma_unmap_single(struct device *dev, dma_addr_t bus_addr, size_t size,
  enum dma_data_direction direction);
```

创建并破坏一次性的流式DMA映射。

```
void dma_sync_single_for_cpu(struct device *dev, dma_handle_t bus_addr, size_t
  size, enum dma_data_direction direction);
void dma_sync_single_for_device(struct device *dev, dma_handle_t bus_addr,
  size_t size, enum dma_data_direction direction);
```

同步具有流映射的缓冲区。如果处理器必须访问流媒体映射时（即设备拥有缓冲区），则必须使用这些功能。

```
#include <asm/scatterlist.h>
struct scatterlist { /* ... */ };
unsigned int sg_dma_len(struct scatterlist *sg);
dma_addr_t sg_dma_address(struct scatterlist *sg);
```

scatterlist结构描述了一个涉及多个缓冲区的I/O操作。宏*sg_dma_address*和*sg_dma_len*可以用来提取总线地址和缓冲长度，以便在实现scatter/chater操作时传递到设备。

```
dma_map_sg(struct device *dev, struct scatterlist *list, int nents,
  enum dma_data_direction direction);
dma_unmap_sg(struct device *dev, struct scatterlist *list, int nents, enum
  dma_data_direction direction);
void dma_sync_sg_for_cpu(struct device *dev, struct scatterlist *sg, int
  nents, enum dma_data_direction direction);
void dma_sync_sg_for_device(struct device *dev, struct scatterlist *sg, int
void dma_sync_sg_for_device(struct device *dev, struct scatterlist *sg, int
  nents, enum dma_data_direction direction);
```

*dma_map_sg*绘制散点/收集操作，*dma_unmap_sg*撤消该映射。如果在映射处于活动状态时必须访问缓冲区，则可以使用*dma_sync_sg_**同步事物。 `/proc/dma`文件，其中包含DMA Conlollers中分配的通道的文本快照。没有显示基于PCI的DMA，因为每个板无需分配DMA控制器中的通道。 `#include <asm/dma.h>`标头定义或原型与DMA相关的所有功能和宏。必须包括使用以下任何符号。

`int request_dma(unsigned int channel, const char *name);`
`void free_dma(unsigned int channel);`
: Access the DMA registry. Registration must be performed before using ISA DMA channels.

`unsigned long claim_dma_lock( );`
`void release_dma_lock(unsigned long flags);`
: Acquire and release the DMA spinlock, which must be held prior to calling the other ISA DMA functions described later in this list. They also disable and reenable interrupts on the local processor.

`void set_dma_mode(unsigned int channel, char mode);`
`void set_dma_addr(unsigned int channel, unsigned int addr);`
`void set_dma_count(unsigned int channel, unsigned int count);`
: Program DMA information in the DMA controller. `addr` is a bus address.

`void disable_dma(unsigned int channel);`
`void enable_dma(unsigned int channel);`
: A DMA channel must be disabled during configuration. These functions change the status of the DMA channel.

`int get_dma_residue(unsigned int channel);`
: If the driver needs to know how a DMA transfer is proceeding, it can call this function, which returns the number of data transfers that are yet to be completed. After successful completion of DMA, the function returns `0`; the value is unpredictable while data is being transferred.

`void clear_dma_ff(unsigned int channel)`
: The DMA flip-flop is used by the controller to transfer 16-bit values by means of two 8-bit operations. It must be cleared before sending any data to the controller.

`int request_dma(unsigned int channel, const char *name);`
`void free_dma(unsigned int channel);`
: 访问DMA注册表。在使用ISA DMA通道之前，必须执行注册。

`unsigned long claim_dma_lock( );`
`void release_dma_lock(unsigned long flags);`
: 获取并释放DMA Spinlock，该锁必须在拨打此列表后面描述的其他ISA DMA函数之前保留。他们还禁用并重新打断本地处理器。

`void set_dma_mode(unsigned int channel, char mode);`
`void set_dma_addr(unsigned int channel, unsigned int addr);`
`void set_dma_count(unsigned int channel, unsigned int count);`
: DMA控制器中的DMA信息。 `addr`是一个总线地址。

`void disable_dma(unsigned int channel);`
`void enable_dma(unsigned int channel);`
: 在配置过程中必须禁用DMA通道。这些功能会改变DMA通道的状态。

`int get_dma_residue(unsigned int channel);`
: 如果驱动程序需要知道如何进行DMA传输，它可以调用此功能，该功能返回尚未完成的数据传输数量。成功完成DMA后，功能返回0;传输数据时，该值是不可预测的。

`void clear_dma_ff(unsigned int channel)`
: 控制器使用DMA触发器通过两个8位操作传输16位值。在将任何数据发送给控制器之前必须清除。