第 15 章

内存映射和DMA

本章深入研究 Linux 内存管理领域,重点是
关于对设备驱动程序编写者有用的技术。许多类型的驱动程序编程需要了解虚拟内存子系统的工作原理。
我们在本章中介绍的材料在我们开始学习时不止一次派上用场
一些更复杂和性能关键的子系统。虚拟内存子系统也是核心 Linux 内核中一个非常有趣的部分,因此值得一看。

本章的材料分为三个部分:

- 第一个涵盖了mmap系统调用的实现,它允许
  将设备内存直接映射到用户进程的地址空间。不是全部
  设备需要mmap支持,但对于某些设备来说,映射设备内存可以产生
  显着的性能改进。
- 然后,我们讨论从另一个方向跨越边界,讨论直接访问用户空间页面。相对较少的驱动程序需要这种能力;在许多情况下,内核在没有驱动程序的情况下执行这种映射

  甚至意识到它。但是了解如何将用户空间内存映射到
  内核(带有get_user_pages)可能很有用。
- 最后一节介绍直接内存访问(DMA) I/O 操作,它为外设提供对系统内存的直接访问。

当然,所有这些技术都需要了解 Linux 内存
管理工作,所以我们从该子系统的概述开始。

Linux 中的内存管理

这不是描述操作系统中的内存管理理论,而是
部分试图确定 Linux 实现的主要特性。虽然
你不需要成为 Linux 虚拟内存大师来实现mmap,一个基本的
对事物如何工作的概述很有用。接下来是相当冗长的描述

内核用来管理内存的数据结构。一旦涵盖了必要的背景,我们就可以开始使用这些结构。

地址类型Linux 当然

是一个虚拟内存系统,这意味着用户程序看到的地址并不直接对应于硬件使用的物理地址。虚拟内存引入了一个间接层,它允许许多好东西。使用虚拟内存,系统上运行的程序可以分配比物理可用内存多得多的内存;实际上,即使是单个进程也可以拥有比系统物理内存更大的虚拟地址空间。虚拟内存还允许程序对进程的地址空间进行一些技巧,包括将程序的内存映射到设备内存。

到目前为止,我们已经讨论了虚拟地址和物理地址,但是一些细节被忽略了。 Linux 系统处理几种类型的地址,每一种都有自己的语义。不幸的是,内核代码并不总是很清楚在每种情况下使用哪种类型的地址,因此程序员必须小心。

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

用户虚拟地址

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 位或 64 位,具体取决于底层硬件架构,并且每个进程都有自己的虚拟地址空间。

物理地址处理器和系

统内存之间使用的地址。物理地址是 32 位或 64 位的数量;在某些情况下,甚至 32 位系统也可以使用更大的物理地址。

总线地址

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,但不一定如此。一些架构可以提供一个 I/O 内存管理单元 (IOMMU),它在总线和主内存之间重新映射地址。 IOMMU 可以通过多种方式使生活更轻松(例如,使分散在内存中的缓冲区与设备相邻),但对 IOMMU 进行编程是设置 DMA 操作时必须执行的额外步骤。当然,总线地址高度依赖于体系结构。

内核逻辑地址这些构成了内

核的正常地址空间。这些地址映射主存储器的某些部分(可能是全部),并且通常被视为物理地址。在大多数架构上,逻辑地址及其相关

物理地址仅相差一个恒定的偏移量。逻辑地址使用硬件的本机指针大小,因此可能无法寻址大量配备的 32 位系统上的所有物理内存。逻辑地址通常存储在unsigned long或void * 类型的变量中。从kmalloc返回的内存具有内核逻辑地址。

内核虚拟地址

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映射。然而,内核虚拟地址不一定具有到表征逻辑地址空间的物理地址的线性、一对一映射。所有逻辑地址都是内核虚拟地址,但许多内核虚拟地址不是逻辑地址。

例如， vmalloc分配的内存有一个虚拟地址（但没有直接的物理映射）。 kmap函数（本章稍后将介绍)也返回虚拟地址。虚拟地址通常存储在指针变量中。

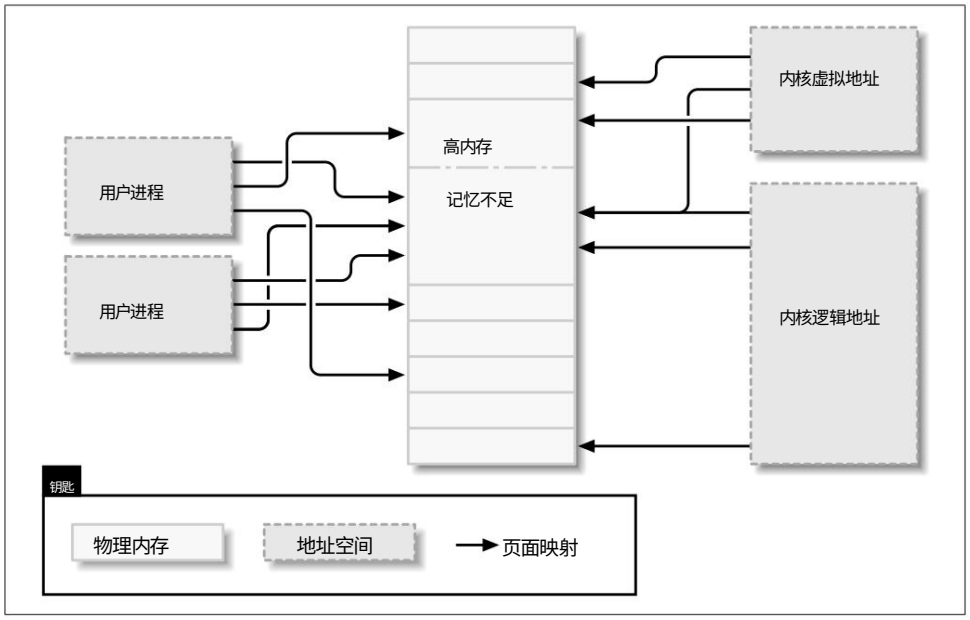

图 15-1。 Linux 中使用的地址类型

如果您有一个逻辑地址,宏__pa( ) （在<asm/page.h> 中定义)返回其关联的物理地址。物理地址可以使用__va( ) 映射回逻辑地址,但仅限于低内存页面。

不同的内核函数需要不同类型的地址。如果定义了不同的 C 类型会很好,这样所需的地址类型是明确的,但我们没有这样的运气。在本章中,我们试图弄清楚在哪里使用了哪些类型的地址。

物理地址和页面物理内存被分成称为页

面的离散单元。系统的大部分内存内部处理都是基于每页完成的。尽管大多数系统当前使用 4096 字节的页面,但页面大小因一种体系结构而异。常量PAGE_SIZE （在<asm/page.h> 中定义)给出了任何给定架构的页面大小。

如果您查看内存地址（虚拟的或物理的）,它可以分为页号和页内的偏移量。例如,如果正在使用 4096 字节的页面,则 12 个最低有效位是偏移量,其余的较高位表示页码。如果丢弃偏移量并将偏移量的其余部分向右移动,则结果称为页框号(PFN)。移位以在页框号和地址之间进行转换是一种相当常见的操作。宏PAGE_SHIFT告诉必须移动多少位才能进行此转换。

高内存和低内存逻辑和内核虚

拟地址之间的差异在配备大量内存的 32 位系统上突出显示。使用 32 位,可以寻址 4 GB 的内存。直到最近,32 位系统上的 Linux 仍被限制为比这要少得多的内存,但是,因为它设置虚拟地址空间的方式。

内核（在 x86 架构上,在默认配置中)在用户空间和内核之间分割了 4 GB 的虚拟地址空间;在两种情况下都使用相同的映射集。典型的拆分将 3 GB 专用于用户空间,1 GB 用于内核空间。* 内核的代码和数据结构必须适合该空间,但内核地址空间的最大消耗者是物理内存的虚拟映射。内核不能直接操作未映射到内核地址空间的内存。换句话说,内核需要它自己的虚拟地址来存放它必须直接接触的任何内存。因此,多年来,内核可以处理的最大物理内存量是可以映射到内核的虚拟地址空间部分的数量,减去空间

* 许多非 x86 架构能够在没有此处描述的内核/用户空间拆分的情况下有效地进行操作,因此它们可以在 32 位系统上使用多达 4 GB 的内核地址空间。但是,当安装了超过 4 GB 的内存时,本节中描述的限制仍然适用于此类系统。

内核代码本身需要。因此,基于 x86 的 Linux 系统最多可以使用略低于 1 GB 的物理内存。

为了应对商业压力,在不破坏 32 位应用程序和系统兼容性的情况下支持更多内存,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 位处理器也可以寻址超过 4 GB 的物理内存。然而,多少内存可以直接映射到逻辑地址的限制仍然存在。只有内存的最低部分（最多 1 或 2 GB,取决于硬件和内核配置)具有逻辑地址;*其余部分（高内存)没有。在访问特定的高内存页面之前,内核必须设置一个显式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在低内存中;高内存往往是为用户空间进程页面保留的。

术语“高内存”可能会让一些人感到困惑,尤其是因为它在 PC 世界中还有其他含义。因此,为了清楚起见,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。在您可能遇到的几乎每个系统上,所有内存都是低内存。

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

在 i386 系统上,低内存和高内存之间的边界通常设置为略低于 1 GB,尽管可以在内核配置时更改该边界。
此边界与原始 PC 上的旧 640 KB 限制没有任何关系,并且其位置不受硬件限制。相反,它是内核本身设置的限制,因为它在内核和用户空间之间分割 32 位地址空间。

在本章中,我们将指出使用高内存的限制。

内存映射和结构页历史上,内核使用逻辑地址来引

用物理内存页。然而,增加对高内存的支持暴露了这种方法的一个明显问题 逻辑地址不适用于高内存。

因此,处理内存的内核函数越来越多地使用指向结构页面的指针（在<linux/mm.h> 中定义） 。该数据结构用于跟踪内核需要了解的有关物理内存的所有信息;

* 2.6 内核（添加了补丁)可以在 x86 硬件上支持“4G/4G”模式,从而以适度的性能成本实现更大的内核和用户虚拟地址空间。

系统上的每个物理页面都有一个结构页面。此结构的一些字段包括以下内容：

atomic_t 计数；
该页面的引用数。当计数下降到0 时，页面返回到空闲列表。无效*虚拟；页面的内核虚拟地址（如果已映射）； NULL，否则。低内存页总是被映射；高内存页通常不是。该字段并非出现在所有架构上；它通常仅在无法轻松计算页面的内核虚拟地址的情况下编译。如果要查看此字段，正确的方法是使用page_address宏，如下所述。

无符号长标志；一组描述
页面状态的位标志。其中包括PG_locked，它指示页面已被锁定在内存中，以及PG_reserved，它完全阻止内存管理系统使用该页面。

struct page中有更多信息，但它是内存管理更深层次的黑魔法的一部分，驱动程序编写者并不关心。

内核维护一个或多个结构页面条目数组，用于跟踪系统上的所有物理内存。在某些系统上，有一个名为mem_map 的数组。
然而，在某些系统上，情况更为复杂。非统一内存访问 (NUMA) 系统和具有广泛不连续物理内存的系统可能具有多个内存映射阵列，因此旨在可移植的代码应尽可能避免直接访问该阵列。幸运的是，使用结构页指针通常很容易，而不用担心它们来自哪里。

定义了一些函数和宏用于在结构页指针和虚拟地址之间进行转换：

结构页 *virt_to_page(void *kaddr);
<asm/page.h>中定义的这个宏接受内核逻辑地址并返回其关联的结构页指针。由于它需要一个逻辑地址，因此它不适用于来自vmalloc的内存或高端内存。

结构页面 *pfn_to_page(int pfn);
返回给定页框号的结构页指针。如有必要，它会在将页框号传递给pfn_to_page之前使用pfn_valid 检查页框号的有效性。无效*page_address（结构页面*page）；

如果存在这样的地址，则返回此页面的内核虚拟地址。对于高端内存，该地址仅在页面已被映射时才存在。这个功能是

在<linux/mm.h> 中定义。在大多数情况下,您希望使用kmap版本而不是page_address。

#include <linux/highmem.h>
void *kmap(struct page *page);
无效kunmap（结构页*页）； kmap

返回系统中任何页面的内核虚拟地址。对于低内存页,它只返回页的逻辑地址;对于高内存页面， kmap在内核地址空间的专用部分创建一个特殊映射。

使用kmap创建的映射应始终使用kunmap 释放;可用的此类映射数量有限,因此最好不要持有它们太久。 kmap调用维护一个计数器,因此如果两个或多个函数都在同一页面上调用 kmap ,那么会发生正确的事情。另请注意,如果没有可用的映射， kmap可以休眠。

#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的一种高性能形式。每个架构都为原子 kmaps 维护一个小槽（专用页表条目)列表； kmap_atomic的调用者必须告诉系统在类型参数中使用这些槽中的哪一个。对驱动程序有意义的唯一插槽是KM_USER0和KM_USER1 （用于直接从用户空间调用运行的代码） ,以及KM_IRQ0和KM_IRQ1 （用于中断处理程序）。请注意,必须以原子方式处理原子 kmap;您的代码在持有一个时无法休眠。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同一个插槽并相互干扰（尽管每个 CPU 都有一组唯一的插槽）。在实践中,原子 kmap 插槽的争用似乎不是问题。

当我们进入示例代码时,我们会看到这些函数的一些用法,在本章后面和后续章节中。

页表在任何现代

系统上,处理器都必须具有将虚拟地址转换为其相应物理地址的机制。这种机制称为页表;它本质上是一个包含虚拟到物理映射和一些相关标志的多级树结构数组。即使在不直接使用此类表的体系结构上,Linux 内核也会维护一组页表。

设备驱动程序通常执行的许多操作可能涉及操作页表。对于驱动程序作者来说幸运的是,2.6 内核消除了直接使用页表的任何需要。因此,我们没有详细描述它们;好奇的读者可能想看看Daniel P. Bovet 和 Marco Cesati (O Reilly) 的Understanding The Linux Ker nel以获得完整的故事。

虚拟内存区

虚拟内存区 (VMA) 是用于管理不同的内核数据结构进程地址空间的区域。一个 VMA 代表一个同质区域进程的虚拟内存:一个连续的虚拟地址范围相同的权限标志并由相同的对象（例如文件或交换空间）。它松散地对应于"段"的概念,尽管它更好被描述为"具有自己属性的内存对象"。进程的内存映射由（至少)以下区域组成:

- 程序可执行代码（通常称为文本)的区域
- 多个数据区域,包括初始化数据（具有显式在执行开始时赋值）,未初始化数据（BSS）,*和程序栈
- 每个活动内存映射一个区域

通过查看/proc/<pid/maps>可以看到进程的内存区域（其中当然， pid被进程 ID 替换）。 /proc/self是/proc/pid的一个特例，因为它总是指当前的进程。例如,这里有几个内存映射（我们添加了斜体的简短注释）:

```
# cat /proc/1/maps查看初始化
08048000-0804E000 R-XP 000000 03:01 64652 0804E          /sbin/初始化文本
000-0804F000 RW-P 00006000 03:01 64652                   /sbin/初始化数据
0804F000-08053000 RWXP 008053000 RWXP                    零映射 BSS
                                                         /lib/ld-2.3.2.so文本
                                                         /lib/ld-2.3.2.so数据
                                                         ld.so 的 BSS
                                                         /lib/tls/libc-2.3.2.so文本
                                                         /lib/tls/libc-2.3.2.so数据
                                                         用于 libc 的 BSS
                                                         堆栈段
00000000000000000000000000000000000000000000000000000000vsyscall 页面0000000000000000000000000000000000000000000000000转机:01 96278 40016000-4

# rsh wolf cat /proc/self/maps #### x86-64（修剪）
00400000-00405000 r-xp 00000000 03:01 1596291 /bin/cat 00504000-00505000 rw-p   /bin/cat   文本
03:01 1596291 00505000-00526000 rwxp 00505000 00:00 0 3252200000-3252214000 r-  00004000   数据
xp 00000000 03:01 1237890 /lib64/ld-2.3.3.so                                                bss

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
7fbffffe000-7fc0000000 rw-p 7fbffffe000 00:00 0 ffffffffff600000-                          堆
ffffffffffe00000 ---p 00000000 00:00 0                                                      系统调用
```

每行中的字段是:

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

* BSS这个名字是一个古老的汇编运算符的历史遗物,意思是"由符号开始的块"。这可执行文件的 BSS 段不存储在磁盘上,内核将零页映射到 BSS 地址范围。

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

开始 结
束
此内存区域的开始和结束虚拟地址。

perm
具有内存区域读取、写入和执行权限的位掩码。该字段描述了允许该进程对属于该区域的页面执行的操作。字段中的最后一个字符是p表示“私人”或s表示“共享”。

offset
内存区域在映射到的文件中的开始位置。偏移量为0表示内存区域的开头对应于文件的开头。

主要
次要
保存已映射文件的设备的主要和次要编号。令人困惑的是，对于设备映射，主要和次要编号是指保存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
文件名（通常是可执行映像）。

vm_area_struct 结构当用户空

间进程调用mmap将设备内存映射到其地址空间时，系统通过创建新的 VMA 来表示该映射来响应。支持mmap （并因此实现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，驱动程序编写者应该至少对 VMA 有一点了解才能支持mmap。

让我们看看struct vm_area_struct （在<linux/mm.h> 中定义）中最重要的字段。设备驱动程序可以在其mmap实现中使用这些字段。
请注意，内核维护 VMA 的列表和树以优化区域查找，并且vm_area_struct的几个字段用于维护此组织。因此，驱动程序不能随意创建 VMA，否则结构会中断。的主要领域

VMA 如下（注意这些字段与我们刚刚看到的/proc输出之间的相似性）：

无符号长 vm_start;无符号
长 vm_end;此 VMA 覆盖的
虚拟地址范围。这些字段是/proc/*/maps 中显示的前两个字段。结构文件 *vm_file;指向与此区域关联的结构文件结构的指针（如果有）。

无符号长 vm_pgoff;文件中
区域的偏移量,以页为单位。映射文件或设备时,这是该区域中映射的第一页的文件位置。

无符号长 vm_flags;一组描述
该区域的标志。设备驱动程序编写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 I/O 区域。除其他外， VM_IO标志可防止该区域包含在进程核心转储中。VM_RESERVED告诉内存管理系统不要尝试换出这个 VMA;它应该在大多数设备映射中设置。

结构 vm_operations_struct *vm_ops;
内核可以调用来操作这个内存区域的一组函数。它的存在表明内存区域是一个内核“对象”，就像我们在本书中一直使用的结构文件一样。

无效 *vm_private_data;
驱动程序可以用来存储自己的信息的字段。

与struct vm_area_struct 一样， vm_operations_struct定义在<linux/mm.h> 中;它包括下面列出的操作。这些操作是处理进程内存需求所需的唯一操作,它们按照声明的顺序列出。本章稍后将实现其中的一些功能。 void (*open)(struct vm_area_struct *vma);内核调用open方法以允许实现 VMA 的子系统初始化该区域。每当对 VMA 进行新引用时（例如,当进程分叉时）,都会调用此方法。一个例外发生在 VMA 第一次由mmap 创建时;在这种情况下,将调用驱动程序的mmap方法。

void (*close)(struct vm_area_struct *vma);当
一个区域被销毁时,内核调用它的关闭操作。请注意,没有与 VMA 相关的使用计数;该区域由使用它的每个进程打开和关闭一次。

struct page *(*nopage)(struct vm_area_struct *vma, unsigned long address, int
*类型）；
当进程尝试访问属于有效 VMA 但当前不在内存中的页面时，将为相关区域调用nopage方法（如果已定义）。该方法可能会在从辅助存储中读取物理页面之后返回该物理页面的结构页面指针。如果没有为该区域定义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);
这种方法允许内核在用户空间访问页面之前将它们“预先设置”到内存中。驱动程序通常不需要实现填充方法。

进程内存映射内存管理难题的最后一

块是进程内存映射结构，它将所有其他数据结构保存在一起。系统中的每个进程（除了一些内核空间帮助线程）都有一个结构 mm_struct （在<linux/sched.h> 中定义），其中包含进程的虚拟内存区域列表、页表和其他各种一些内存管理管理信息，以及一个信号量(mmap_sem)和一个自旋锁(page_table_lock)。指向该结构的指针在任务结构中找到；在驱动程序需要访问它的极少数情况下，通常的方法是使用current->mm。注意内存管理结构可以在进程之间共享；例如，Linux 的线程实现就是以这种方式工作的。

我们对 Linux 内存管理数据结构的概述到此结束。有了这些，我们现在可以继续执行mmap系统调用。

mmap 设备操作

内存映射是现代 Unix 系统最有趣的特性之一。就驱动程序而言，可以实现内存映射来为用户程序提供对设备内存的直接访问。

通过查看 X Window System 服务器的虚拟内存区域的子集，可以看到mmap使用的明确示例：

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

X 服务器的 VMA 的完整列表很长,但这里大部分条目并不感兴趣。然而,我们确实看到了/dev/mem 的四个独立映射,它们提供了一些关于 X 服务器如何与视频卡一起工作的见解。第一个映射位于a0000,这是 640 KB ISA 孔中视频 RAM 的标准位置。再往下,我们在e8000000 处看到一个大映射,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:视频 RAM 区域
000c0000-000ccfff:视频 ROM
000d1000-000d1fff:适配器 ROM
000f0000-000fffff:系统 ROM d7f00000-
f7efffff:PCI 总线 #01
  e8000000-efffffff : 0000:01:00.0
fc700000-fccfffff : PCI 总线 #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内存相关联。每当程序在分配的地址范围内读取或写入时,它实际上是在访问设备。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。对于像这样的性能关键型应用程序,直接访问会产生很大的不同。

您可能会怀疑,并非每个设备都适合mmap抽象。例如,对于串行端口和其他面向流的设备来说,这是没有意义的。 mmap的另一个限制是映射是PAGE_SIZE粒度的。内核只能在页表级别管理虚拟地址;因此,映射区域必须是PAGE_SIZE的倍数,并且必须位于从 PAGE_SIZE 倍数的地址开始的物理内存中。如果区域的大小不是页面大小的倍数,内核会通过使区域稍大来强制大小粒度。

这些限制对驱动程序来说并不是一个很大的限制,因为访问设备的程序无论如何都是依赖于设备的。由于程序必须了解设备的工作原理,因此程序员不会因需要查看页面对齐等细节而过度烦恼。在某些非 x86 平台上使用 ISA 设备时存在更大的限制,因为它们的 ISA 硬件视图可能不连续。例如,一些 Alpha 计算机将 ISA 内存视为一组分散的 8 位、16 位或 32 位项目,没有直接映射。在这种情况下,您根本不能使用mmap 。

无法执行 ISA 地址到 Alpha 地址的直接映射是由于两个系统的数据传输规范不兼容。早期的 Alpha 处理器只能发出 32 位和 64 位内存访问,而 ISA 只能进行 8 位和 16 位传输,并且没有办法透明地将一个协议映射到另一个协议。

在可行的情况下使用mmap有很多好处。例如,我们已经研究过 X 服务器,它往返传输大量数据

显存;与lseek/write实现相比,将图形显示映射到用户空间显着提高了吞吐量。另一个典型的例子是控制 PCI 设备的程序。大多数 PCI 外设将它们的控制寄存器映射到内存地址,高性能应用程序可能更喜欢直接访问寄存器,而不是反复调用ioctl来完成工作。

mmap方法是file_operations结构的一部分,在发出 mmap 系统调用时调用。使用mmap,内核在调用实际方法之前执行了大量工作,因此,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,内核在调用方法之前不会做太多事情。

系统调用声明如下（如mmap(2)手册页中所述）：

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

另一方面,文件操作声明为：

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

方法中的filp参数与第 3 章介绍的相同,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,驱动程序只需为地址范围构建合适的页表,如果需要,用一组新的操作替换vma->vm_ops 。

有两种构建页表的方法:使用名为remap_pfn_range的函数一次性完成,或者通过nopage VMA 方法一次完成一个页面。
每种方法都有其优点和局限性。我们从更简单的“一次性”方法开始。从那里,我们添加了现实世界实施所需的复杂性。

使用 remap_pfn_range

构建新页表以映射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);
```

函数返回的值是通常的0或负错误代码。让我们看一下函数参数的确切含义：

虚拟机

页范围被映射到的虚拟内存区域。

virt_addr

应该开始重新映射的用户虚拟地址。该函数为virt_addr和virt_addr+size之间的虚拟地址范围构建页表。

pfn

对应于虚拟地址应该映射到的物理地址的页帧号。页帧号只是由PAGE_SHIFT位右移的物理地址。对于大多数用途，VMA 结构的vm_pgoff字段正好包含您需要的值。该函数影响从(pfn<<PAGE_SHIFT)到(pfn<<PAGE_SHIFT)+size 的物理地址。

size

被重新映射的区域的尺寸（以字节为单位）。

prot

为新 VMA 请求的“保护”。驱动程序可以（并且应该）使用在vma->vm_page_prot 中找到的值。

remap_pfn_range的参数非常简单，当调用mmap方法时，它们中的大部分已经在 VMA 中提供给您。但是，您可能想知道为什么有两个函数。第一个(remap_pfn_range)用于 pfn 指实际系统 RAM 的情况，而io_remap_page_range应在phys_addr指向 I/O 内存时使用。实际上，这两个函数在除 SPARC 之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都使用 remap_pfn_range 。但是，为了编写可移植驱动程序，您应该使用适合您特定情况的 remap_pfn_range变体。

另一种复杂情况与缓存有关：通常，对设备内存的引用不应由处理器缓存。通常系统 BIOS 会正确设置，但也可以通过保护字段禁用特定 VMA 的缓存。不幸的是，在此级别禁用缓存高度依赖于处理器。好奇的读者可能希望查看drivers/char/mem.c中的pgprot_noncached函数以了解其中涉及的内容。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

要将设备内存简单、线性地映射到用户地址空间，那么 remap_pfn_range几乎是您真正需要完成的工作。下面的代码是

源自drivers/char/mem.c ,并展示了如何在一个名为simple （Simple Implementation Mapping Pages with Little Enthusiasm)的典型模块中执行此任务：

```
静态 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)) 返回-EAGAIN；


    vma->vm_ops = &simple_remap_vm_ops;
    simple_vma_open(vma);返回0；

}
```

如您所见,重新映射内存只需调用remap_pfn_range以创建必要的页表。

添加 VMA 操作正如我们所见，

vm_area_struct结构包含一组可应用于 VMA 的操作。现在我们着眼于以一种简单的方式提供这些操作。
特别是,我们为我们的 VMA 提供打开和关闭操作。每当进程打开或关闭 VMA 时,都会调用这些操作;特别是,只要进程分叉并创建对 VMA 的新引用,就会调用open方法。

除了内核执行的处理之外,还调用了打开和关闭VMA 方法,因此它们不需要重新实现在那里完成的任何工作。它们的存在是驱动程序进行他们可能需要的任何额外处理的一种方式。

事实证明,像 simple 这样的简单驱动程序不需要特别做任何额外的处理。所以我们创建了open和close方法,它们在系统日志中打印一条消息,通知世界它们已被调用。不是特别有用,但它确实允许我们展示如何提供这些方法,并查看它们何时被调用。

为此,我们使用调用printk的操作覆盖默认的vma->vm_ops：

```
无效 simple_vma_open（结构 vm_area_struct *vma){

    printk(KERN_NOTICE    Simple VMA open, virt %lx, phys %lx\n    ,
           vma->vm_start, vma->vm_pgoff << PAGE_SHIFT);
}

无效 simple_vma_close（结构 vm_area_struct *vma){

    printk(KERN_NOTICE    简单的 VMA 关闭。\n    );
}

静态结构 vm_operations_struct simple_remap_vm_ops = { .open =
    simple_vma_open, .close = simple_vma_close,

};
```

要使这些操作对特定映射有效,有必要在相关 VMA的vm_ops字段中存储指向 simple_remap_vm_ops的指针。这通常在mmap方法中完成。如果您返回到 simple_remap_mmap 示例,您会看到以下代码行:

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

请注意对simple_vma_open 的显式调用。由于在初始mmap上未调用open方法,因此如果我们希望它运行,我们必须显式调用它。

使用 nopage 映射内存尽管remap_pfn_range

对于许多（如果不是大多数)驱动程序mmap实现都适用,但有时需要更灵活一些。在这种情况下,可能会调用使用nopage VMA 方法的实现。

nopage方法有用的一种情况可以通过mremap系统调用来实现,应用程序使用它来更改映射区域的边界地址。碰巧的是,当映射的 VMA 被mremap 更改时,内核不会直接通知驱动程序。如果 VMA 的大小减小,内核可以在不告诉驱动程序的情况下悄悄地清除不需要的页面。相反,如果 VMA 被扩展,驱动程序最终会通过调用nopage来发现何时必须为新页面设置映射 ping,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,我们展示了简单设备的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);
```

当用户进程试图访问 VMA 中内存中不存在的页面时,将调用关联的nopage函数。 address参数包含导致错误的虚拟地址,向下舍入到页面的开头。 nopage函数必须定位并返回指向用户想要的页面的结构页面指针。此函数还必须注意通过调用get_page宏来增加它返回的页面的使用计数:

```
get_page(结构页面 *pageptr);
```

此步骤对于保持映射页面上的引用计数正确是必要的。内核为每一页维护这个计数;当计数变为0时,内核知道该页面可能被放置在空闲列表中。当 VMA 未映射时,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向区域添加页面时不增加计数,则使用计数会提前变为0 ,并且系统的完整性会受到损害。

nopage方法还应该将故障类型存储在type参数指向的位置 但前提是该参数不为NULL 。在设备驱动程序中,类型的正确值总是VM_FAULT_MINOR。

如果您使用的是nopage,则在调用mmap时通常需要做的工作很少;我们的版本如下所示:

```
静态 int simple_nopage_mmap(struct file *filp, struct vm_area_struct *vma) {

    无符号长偏移 = 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);返回0;

}
```

mmap要做的主要事情是用我们自己的操作替换默认的(NULL) vm_ops指针。 nopage方法然后负责一次“重新映射”一页并返回其结构页面结构的地址。因为我们在这里只是在物理内存上实现一个窗口,所以重新映射步骤很简单:我们只需要定位并返回一个指向所需地址的结构页面的指针。我们的nopage方法如下所示:

```
struct page *simple_vma_nopage(struct vm_area_struct *vma,
                               unsigned long address, int *type)
{
    结构页面 *pageptr;无符号
    长偏移 = vma->vm_pgoff << PAGE_SHIFT;无符号长 physaddr
    = 地址 - vma->vm_start + 偏移量; unsigned long pageframe =
    physaddr >> PAGE_SHIFT;

    if (!pfn_valid(pageframe)) 返回
        NOPAGE_SIGBUS;
    pageptr = pfn_to_page(pageframe);
    get_page(pageptr); if (type) *type =
    VM_MINOR_FAULT;返回页面指针;


}
```

因为,再一次,我们只是在这里映射主内存,所以nopage函数只需要为错误地址找到正确的结构页面并增加它的引用计数。因此,所需的事件序列是计算所需的物理地址,并通过右移PAGE_SHIFT位将其转换为页帧号。

由于用户空间可以给我们任何它喜欢的地址,我们必须确保我们有一个有效的页框; pfn_valid函数为我们做这件事。如果地址超出范围,我们返回NOPAGE_SIGBUS,这会导致将总线信号传递给调用进程。

否则， pfn_to_page获取必要的struct page指针;我们可以增加它的引用计数（通过调用get_page)并返回它。

nopage方法通常返回一个指向结构页面的指针。如果由于某种原因无法返回正常页面（例如,请求的地址超出了设备的内存区域）,则可以返回NOPAGE_SIGBUS以指示错误;这就是上面的简单代码所做的。 nopage也可以返回NOPAGE_OOM以指示由于资源限制导致的失败。

请注意,此实现适用于 ISA 内存区域,但不适用于 PCI 总线上的内存区域。 PCI 内存映射在最高系统内存之上,系统内存映射中没有这些地址的条目。因为没有struct page可以返回指针,所以在这些情况下不能使用nopage ;您必须改用remap_pfn_range 。

如果nopage方法保留为NULL,则处理页面错误的内核代码会将零页映射到错误的虚拟地址。零页是读为0的写时复制页,例如用于映射 BSS 段。任何引用零页面的进程都可以准确地看到:一个充满零的页面。如果进程写入页面,它最终会修改私有副本。因此,如果一个进程通过调用mremap扩展了一个映射区域,并且驱动程序没有实现nopage,则该进程以零填充内存结束,而不是分段错误。

重新映射特定的 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;他们将物理地址重新映射到用户空间。然而,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,而不是所有内存。为了将整个内存范围的一个子集映射到用户空间,驱动程序只需要使用偏移量。以下是驱动程序映射simple_region_size字节区域的技巧,从物理地址simple_region_start开始（应该是页面对齐的）：

```
无符号长关 = vma->vm_pgoff << PAGE_SHIFT;无符号长物理
= simple_region_start + off； unsigned long vsize = vma-
>vm_end - vma->vm_start;无符号长 psize = simple_region_size
- 关闭；

如果（vsize > psize)返
      回-EINVAL； /* 跨度太高 */
remap_pfn_range（vma,vma_>vm_start,物理,vsize,vma->vm_page_prot）；
```

除了计算偏移量之外,此代码还引入了一个检查,当程序尝试映射的内存多于目标设备的 I/O 区域中可用的内存时,该检查会报告错误。在这段代码中， psize是指定偏移量后剩下的物理 I/O 大小， vsize是请求的虚拟内存大小;该函数拒绝映射超出允许内存范围的地址。

请注意,用户进程始终可以使用mremap来扩展其映射,可能超出物理设备区域的末尾。如果您的驱动程序未能定义nopage方法,则永远不会通知此扩展,并且附加区域映射到零页。作为驱动程序编写者,您可能希望防止这种行为;将零页映射到您所在区域的末尾并不是一件明显的坏事,但程序员不太可能希望这种情况发生。

防止映射扩展的最简单方法是实现一个简单的nopage方法,该方法总是会导致将总线信号发送到故障进程。
这样的方法看起来像这样:

```
struct page *simple_nopage(struct vm_area_struct *vma,
                           unsigned long address, int *type); { 返
回 NOPAGE_SIGBUS;  /* 发送一个 SIGBUS */}
```

正如我们所看到的,只有当进程取消引用位于已知 VMA 内但当前没有有效页表条目的地址时,才会调用nopage方法。如果我们使用remap_pfn_range来映射整个设备区域,则此处显示的nopage方法仅针对该区域之外的引用调用。因此,它可以安全地返回NOPAGE_SIGBUS以发出错误信号。当然,更彻底的nopage实现可以检查故障地址是否在设备区域内,如果是,则执行重新映射。然而,再一次, nopage不适用于 PCI 内存区域,因此 PCI 映射的扩展是不可能的。

重新映射 RAM

remap_pfn_range的一个有趣的限制是它只允许访问保留页面和物理内存顶部之上的物理地址。在 Linux 中,物理地址页在内存映射中被标记为“保留”,表示它不可用于内存管理。例如,在 PC 上,640 KB 和 1 MB 之间的范围被标记为保留,托管内核代码本身的页面也是如此。保留页被锁定在内存中,并且是唯一可以安全映射到用户空间的页;这种限制是系统稳定性的基本要求。

因此, remap_pfn_range不允许您重新映射常规地址,其中包括您通过调用get_free_page 获得的地址。相反,它映射在零页中。一切似乎都正常工作,除了进程看到私有的、零填充的页面,而不是它希望的重新映射的 RAM。尽管如此,该函数完成了大多数硬件驱动程序需要它做的所有事情,因为它可以重新映射高 PCI 缓冲区和 ISA 内存。

remap_pfn_range的限制可以通过运行mapper来看出,mapper是 O Reilly 的 FTP 站点上提供的文件中misc-progs中的示例程序之一。 mapper是一个简单的工具,可以用来快速测试mmap系统调用;它映射由命令行选项指定的文件的只读部分,并将映射的区域转储到标准输出。例如,以下会话显示/dev/mem 没有

映射位于地址 64 KB 的物理页面相反,我们看到一个全零的页面（本示例中的主机是 PC,但在其他平台上结果将相同）： morgana.root# ./mapper /开发/内存 0x10000 0x1000 | od -Ax -t x1 将“/dev/mem”从 65536 映射到 69632

```
000000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
*
001000
```

remap_pfn_range无法处理 RAM 表明基于内存的设备（如scull)无法轻松实现mmap,因为它的设备内存是传统的 RAM,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简单的解决方法。它使用我们之前看到的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 一次处理一个页面错误。示例实现是第 8 章中介绍的scullp模块的一部分。scullp是一个面向页面的字符设备。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了“Linux 中的内存管理”一节中介绍的一些概念。

在检查代码之前,让我们看看影响scullp中mmap实现的设计选择：

- 只要设备被映射， scullp就不会释放设备内存。这是一个政策问题而不是要求,它与scull和类似设备的行为不同,后者在打开写入时会被截断为长度为0 。拒绝释放映射的scullp设备允许进程覆盖由另一个进程主动映射的区域,因此您可以测试并查看进程和设备内存如何交互。为避免释放映射设备,驱动程序必须保持活动映射的计数;设备结构中的vmas字段用于此目的。

- 仅当scullp order参数（在模块加载时设置)为0 时才执行内存映射。该参数控制如何调用 __get_free_pages （参见第8 章中的“get_free_page 和Friends”一节）。零阶限制（强制页面一次分配一个,而不是在更大的组中)由__get_free_pages的内部决定,__get_free_pages是scullp使用的分配函数。为了最大化分配性能,Linux内核为每个分配顺序维护了一个空闲页面列表,只有集群中第一页的引用计数增加get_free_pages和减少free_pages。如果分配顺序大于零,则对scullp设备禁用mmap方法,因为nopage处理单个页面而不是页面集群。雕刻

根本不知道如何正确管理属于高阶分配的页面的引用计数。 （如果您需要复习scullp和内存分配顺序值,请返回第 8 章中的“使用整页的 scull:scullp”部分。）

零阶限制主要是为了保持代码简单。可以通过使用页面的使用计数来正确实现多页面分配的 mmap ,但这只会增加示例的复杂性,而不会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;它还需要访问内存映射来调整页面使用计数。

scullp_mmap的这个实现很短,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) {

    结构 inode *inode = filp->f_dentry->d_inode;

    /* 如果 order 不为 0,则拒绝映射 */ if
    (scullp_devices[iminor(inode)].order) return
        -ENODEV;

    /* 在这里不要做任何事情：“nopage”将填补漏洞 */ vma->vm_ops =
    &scullp_vm_ops; vma->vm_flags |= VM_RESERVED; vma->vm_private_data
    = filp->private_data; scullp_vma_open(vma);返回0；


}
```

if语句的目的是避免映射分配顺序不为 0的设备。scullp的操作存储在vm_ops字段中,指向设备结构的指针存储在vm_private_data字段中。最后,调用vm_ops->open来更新设备的活动映射计数。 open和close只是跟踪映射计数,定义如下：

```
void scullp_vma_open(struct vm_area_struct *vma) {

    结构 scullp_dev *dev = vma->vm_private_data;

    开发->vmas++；
}

无效 scullp_vma_close（结构 vm_area_struct *vma){

    结构 scullp_dev *dev = vma->vm_private_data;

    开发->vms--；
}
```

然后大部分工作由nopage 执行。在scullp实现中， nopage的地址参数用于计算进入设备的偏移量;然后使用偏移量在scullp内存树中查找正确的页面： struct page *scullp_vma_nopage(struct vm_area_struct *vma, unsigned long address, int *type)

```
{
    无符号长偏移量;结构
    scullp_dev *ptr, *dev = vma->vm_private_data;结构页*页=
    NOPAGE_SIGBUS;无效 *pageptr = NULL; /* 默认为“缺失” */

    向下（&dev->sem）；
    偏移量 = (地址 - vma->vm_start) + (vma->vm_pgoff << PAGE_SHIFT); if (offset >=
    dev->size) goto out; /* 超出范围 */

    /*
     * 现在从列表中检索 scullp 设备,然后是页面。
     * 如果设备有空洞,则进程在访问空洞时会收到 SIGBUS。 */
     *

    偏移量 >>= PAGE_SHIFT； /* offset 是页数 */ for (ptr = dev; ptr &&
    offset >= dev->qset;) { ptr = ptr->next;偏移量-=开发->qset；

    } if (ptr && ptr->data) pageptr = ptr->data[offset]; if (!pageptr)
    退出； /* 空洞或文件结尾 */ page = virt_to_page(pageptr);

    /* 明白了,现在增加计数 */ get_page(page); if
    (type) *type = VM_FAULT_MINOR;

  出去：
    向上（&dev-
    >sem）;返回页面；
}
```

scullp使用通过get_free_pages 获得的内存。该内存是使用逻辑地址寻址的,因此scullp_nopage所要做的就是调用virt_to_page 来获取struct page指针。

scullp设备现在按预期工作,正如您在映射器实用程序的此示例输出中看到的那样。在这里,我们将/dev的目录列表（很长)发送到scullp设备,然后使用mapper实用程序通过mmap查看该列表的各个部分：

```
morgana% ls -l /dev > /dev/scullp
morgana% ./mapper /dev/scullp 0 140映射
 “/dev/scullp”从 0 (0x00000000) 到 140 (0x0000008c) 总计 232

crw--------    1根    根    15 年 9 月 10 日 10 日 07:40
```

```
crw-r--r--    1根    根          175 年 9 月 10 日 15 日 07:40
morgana% ./mapper /dev/scullp 8192 200
将“/dev/scullp”从 8192 (0x00002000) 映射到 8392 (0x000020c8)
d0h1494
brw-rw----    1根    软盘 软盘    2015 年 9 月 92 日 07:40 fd0h1660
brw-rw----    1根                 2015 年 9 月 20 日 07:40 fd0h360
brw-rw----    1根                 2015 年 9 月 12 日 07:40 fd0H360
```

重新映射内核虚拟地址

尽管很少需要,但看看驱动程序如何映射内核还是很有趣的
使用mmap到用户空间的虚拟地址。记住,真正的内核虚拟地址是
由诸如vmalloc 之类的函数返回的地址,即映射的虚拟地址
在内核页表中。本节中的代码取自scullv,即
像scullp一样工作但通过vmalloc分配其存储的模块。

大多数scullv实现就像我们刚刚看到的scullp 一样,除了
无需检查控制内存分配的order参数。
这样做的原因是vmalloc一次分配一个页面,因为单页分配比多页分配更有可能成功。因此,分配
顺序问题不适用于vmalloced空间。

除此之外,使用的nopage实现之间只有一个区别
scullp和scullv。请记住， scullp一旦找到感兴趣的页面,就会获得
与virt_to_page对应的struct page指针。该功能不起作用
但是,使用内核虚拟地址。相反,您必须使用vmalloc_to_page。所以
nopage的scullv版本的最后一部分如下所示：

```
    /*
     * scullv 查找后，“page”现在是页面的地址
     * 当前进程需要。由于它是一个vmalloc地址，
     * 把它变成一个结构页面。
     */
    page = vmalloc_to_page(pageptr);

    /* 明白了,现在增加计数 */
    获取页面（页面）；
    如果（类型）
        *类型 = VM_FAULT_MINOR;
出去：
    向上（&dev->sem）；
    返回页面；
```

基于此讨论,您可能还希望映射ioremap返回的地址
到用户空间。然而,那将是一个错误。 ioremap的地址是特殊的
并且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 内存区域重新映射到用户空间。

执行直接 I/O

大多数 I/O 操作都是通过内核缓冲的。内核空间缓冲区的使用允许用户空间和实际设备之间有一定程度的分离;这种分离可以使编程更容易,并且还可以在许多情况下产生性能优势。然而,在某些情况下,直接对用户空间缓冲区执行 I/O 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,通过内核空间直接传输数据而不需要额外的副本可以加快速度。

在 2.6 内核中使用直接 I/O 的一个示例是 SCSI 磁带驱动程序。流式磁带可以通过系统传递大量数据,而磁带传输通常是面向记录的,因此在内核中缓冲数据几乎没有什么好处。因此,当条件合适时（例如,用户空间缓冲区是页面对齐的）,SCSI 磁带驱动程序执行其 I/O 而不复制数据。

也就是说,重要的是要认识到直接 I/O 并不总能提供人们可能期望的性能提升。设置直接 I/O 的开销（这涉及到相关的用户页面的故障和锁定)可能是显着的,并且缓冲 I/O 的好处会丢失。例如,使用直接 I/O 要求write系统调用同步操作;否则应用程序不知道何时可以重用其 I/O 缓冲区。在每次写入完成之前停止应用程序会减慢速度,这就是使用直接 I/O 的应用程序通常也使用异步 I/O 操作的原因。

无论如何,这个故事的真正寓意是在 char 驱动程序中实现直接 I/O 通常是不必要的,而且可能会造成伤害。只有当您确定缓冲 I/O 的开销确实会减慢速度时,您才应该采取该步骤。另请注意,块和网络驱动程序根本不需要担心实现直接 I/O;在这两种情况下,内核中的高级代码都会在指示时设置并使用直接 I/O,而驱动程序级代码甚至不需要知道正在执行直接 I/O。

在 2.6 内核中实现直接 I/O 的关键是一个名为get_user_pages 的函数,它在<linux/mm.h>中声明,原型如下:

```
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);
```

这个函数有几个参数：

tsk

指向执行 I/O 的任务的指针；它的主要目的是告诉内核在设置缓冲区时发生的任何页面错误应该由谁负责。

此参数几乎总是作为当前参数传递。 mm指向描述要映

射的地址空间的内存管理结构的指针。 mm_struct结构是将进程的虚拟地址空间的所有部分(VMA) 联系在一起的部分。对于驱动程序使用，此参数应始终为current->mm。

开始

只要

start是用户空间缓冲区的（页对齐）地址， len是缓冲区的页长度。

写

force

如果write不为零，则映射页面以进行写访问（当然，这意味着该用户空间正在执行读操作）。force标志告诉get_user_pages覆盖给定页面上的保护以提供请求的访问权限；驱动程序应始终在此处传递0 。

页面

虚拟机

输出参数。成功完成后，页面包含指向描述用户空间缓冲区的struct page结构的指针列表，而vmas包含指向关联 VMA 的指针。显然，参数应该指向能够保存至少len指针的数组。任何一个参数都可以为NULL，但您至少需要struct page指针才能对缓冲区进行实际操作。

get_user_pages是一个低级内存管理函数，具有适当复杂的接口。它还要求在调用之前以读取模式获取地址空间的 mmap 读取器/写入器信号量。因此，对get_user_pages的调用通常如下所示：

```
down_read(&current->mm-
>mmap_sem);结果= get_user_pages（当前，当前->毫
米，...）； up_read(&current->mm->mmap_sem);
```

返回值是实际映射的页数，可能小于请求的数量（但大于零）。

成功完成后，调用者有一个pages数组指向用户空间缓冲区，该缓冲区被锁定到内存中。要直接对缓冲区进行操作，内核空间代码必须使用kmap或kmap_atomic将每个struct page指针转换为内核虚拟地址。然而，通常，直接 I/O 合理的设备使用 DMA 操作，因此您的驱动程序可能希望创建分散/聚集

结构页指针数组中的列表。我们将在“分散/聚集映射”部分讨论如何做到这一点。

一旦您的直接 I/O 操作完成,您必须释放用户页面。但是,在这样做之前,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,并释放它们而不将它们写出到后备存储。因此,如果您更改了页面(响应用户空间读取请求),则必须通过调用以下方法将每个受影响的页面标记为脏:

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

(这个宏在<linux/page-flags.h> 中定义)。大多数执行此操作的代码首先检查以确保该页面不在内存映射的保留部分中,该保留部分永远不会被换出。因此,代码通常如下所示:

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

面);由于用户空间内存通常不会标记为保留,因此严格来说不需要进行此检查,但是当您深入到内存管理子系统中时,最好彻底和小心。

无论页面是否已更改,它们都必须从页面缓存中释放,否则它们将永远留在那里。使用的调用是:

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

当然,这个调用应该在页面被标记为脏之后进行,如果需要的话。

异步 I/O添加到 2.6 内核

的新特性之一是异步 I/O功能。异步 I/O 允许用户空间启动操作而无需等待操作完成;因此,应用程序可以在其 I/O 运行时进行其他处理。一个复杂的高性能应用程序也可以使用异步 I/O 来同时进行多个操作。

异步 I/O 的实现是可选的,驱动作者很少打扰;大多数设备都无法从此功能中受益。正如我们将在接下来的章节中看到的,块和网络驱动程序在任何时候都是完全异步的,因此只有字符驱动程序是显式异步 I/O 支持的候选者。如果有充分的理由在任何给定时间有多个 I/O 操作未完成,则 char 设备可以从这种支持中受益。一个很好的例子是流式磁带驱动器,如果 I/O 操作没有足够快地到达,驱动器可能会停止并显着减速。试图从流驱动器中获得最佳性能的应用程序可以使用异步 I/O 在任何给定时间准备好多个操作。

对于需要实现异步 I/O 的少数驱动程序作者,我们将简要介绍其工作原理。我们将在本章中介绍异步 I/O,因为它的实现几乎总是涉及直接 I/O 操作（如果您在内核中缓冲数据,您通常可以实现异步行为而不会增加用户空间的复杂性）。

支持异步 I/O 的驱动程序应该包括<linux/aio.h>。异步I/O的实现有3种file_operations方法：
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操作只对文件系统代码感兴趣,所以我们在这里不再讨论它。另外两个， aio_read和aio_write,看起来很像常规的读写方法,但有几个例外。一是offset参数是传值的;异步操作从不改变文件位置,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,我们稍后会介绍。

aio_read和aio_write方法的目的是启动读取或写入操作,该操作在返回时可能已完成,也可能未完成。如果可以立即完成操作,则该方法应该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,如果您的驱动程序有一个名为my_read的读取方法,则以下aio_read方法是完全正确的（尽管毫无意义）：

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

请注意,结构文件指针位于kiocb结构的ki_filp字段中。

如果您支持异步 I/O,您必须了解内核有时会创建“同步 IOCB”这一事实。这些本质上是必须同步执行的异步操作。人们可能很想知道为什么会这样做,但最好只是按照内核的要求去做。同步操作标记在 IOCB 中;您的驱动程序应通过以下方式查询该状态：

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

如果此函数返回一个非零值,您的驱动程序必须同步执行该操作。

然而,最后,所有这些结构的重点是启用异步操作。如果您的驱动程序能够启动操作（或者简单地说,将其排队到将来可以执行的某个时间）,它必须做两件事:记住它的所有内容

需要了解操作,并将-EIOCBQUEUED返回给调用者。记忆操作信息包括安排对用户空间缓冲区的访问;一旦返回,在调用进程的上下文中运行时,您将不再有机会访问该缓冲区。一般来说,这意味着您可能必须设置直接内核映射（使用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“稍后”到来时,您的驱动程序必须通知内核操作已完成。这是通过调用aio_complete 来完成的:

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

在这里， iocb是最初传递给您的同一个 IOCB,而res是操作的通常结果状态。 res2是返回给用户空间的第二个结果码;大多数异步 I/O 实现将res2传递为0。一旦调用aio_complete,就不应再次触摸 IOCB 或用户缓冲区。

异步 I/O 示例示例源代码中面向

页面的scullp驱动程序实现了异步 I/O。
实现很简单,但足以说明异步操作的结构。

aio_read和aio_write方法实际上并没有做太多事情:

```
静态 ssize_t scullp_aio_read(struct kiocb *iocb, char *buf, size_t count, loff_t pos)

{
    返回 scullp_defer_op(0, iocb, buf, count, pos);
}

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

这些方法只是调用一个通用函数:

```
结构 async_work { 结
    构 kiocb *iocb;整数结
    果;结构工作_结构工作;

};

静态 int scullp_defer_op(int write, struct kiocb *iocb, char *buf, size_t count,
            loff_t pos)
{
    结构 async_work *stuff;整数结
    果;
```

```
/* 在我们可以访问缓冲区时立即复制 */ if (write) result
= scullp_write(iocb->ki_filp, buf, count, &pos);否则
      结果 = scullp_read(iocb->ki_filp, buf, count, &pos);


/* 如果这是一个同步 IOCB,我们现在返回我们的状态。 */ if
(is_sync_kiocb(iocb)) 返回结果;


/* 否则将完成延迟几毫秒。 */ stuff = kmalloc (sizeof (*stuff),
GFP_KERNEL); if (stuff = = NULL) 返回结果； /* 没有内存,现在就完成 */
stuff->iocb = iocb;东西->结果=结果； INIT_WORK(&stuff->work,
      scullp_do_deferred_op, stuff); schedule_delayed_work(&stuff-
>work, HZ/100);返回-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。我们选择从一开始就复制数据来保持简单。然后调用is_sync_kiocb ,看这个操作是否必须同步完成;如果是,则返回结果状态,我们就完成了。否则,我们会在一个小结构中记住相关信息,通过工作队列安排“完成”,并返回-EIOCBQUEUED。此时,控制权返回给用户

空间。

稍后,工作队列执行我们的完成功能:

```
静态 void scullp_do_deferred_op(void *p) {

    struct async_work *stuff = (struct async_work *) p;
    aio_complete(stuff->iocb, stuff->result, 0); kfree（东西）；

}
```

在这里,只需使用我们保存的信息调用aio_complete 即可。当然,真正的驱动程序的异步 I/O 实现要复杂一些,但它遵循这种结构。

直接内存访问

直接内存访问或 DMA 是完成我们对内存问题的概述的高级主题。 DMA 是一种硬件机制,它允许外围组件直接将其 I/O 数据传入和传出主内存,而无需涉及系统处理器。使用这种机制可以大大增加进出设备的吞吐量,因为消除了大量的计算开销。

DMA 数据传输概述

在介绍编程细节之前,让我们回顾一下 DMA 传输是如何发生的,只考虑输入传输以简化讨论。

数据传输可以通过两种方式触发:软件请求数据（通过读取等功能)或硬件异步将数据推送到系统。

在第一种情况下,所涉及的步骤可以总结如下:

1. 当进程调用read 时,驱动方法分配一个 DMA 缓冲区并指示硬件将其数据传输到该缓冲区。进程进入休眠状态。

2. 硬件将数据写入 DMA 缓冲区,并在到达时引发中断
完毕。

3. 中断处理程序获取输入数据,确认中断,唤醒进程,现在可以读取数据。

第二种情况出现在异步使用 DMA 时。例如,即使没有人在读取数据,数据采集设备也会继续推送数据,就会发生这种情况。在这种情况下,驱动程序应该维护一个缓冲区,以便后续的读取调用将所有累积的数据返回到用户空间。这种转移涉及的步骤略有不同:

1. 硬件产生一个中断来宣布新数据到达。
2. 中断处理程序分配一个缓冲区并告诉硬件传输到哪里
它的数据。

3.外围设备将数据写入缓冲区并引发另一个中断
完成后。

4. 处理程序分派新数据,唤醒任何相关进程,并负责内务处理。

异步方法的一种变体经常出现在网卡中。这些卡通常希望看到在与处理器共享的内存中建立的循环缓冲区（通常称为DMAring 缓冲区） 。每个传入的数据包都放置在环中的下一个可用缓冲区中,并发出中断信号。然后驱动程序将网络数据包传递给内核的其余部分,并在环中放置一个新的 DMA 缓冲区。

所有这些情况下的处理步骤都强调有效的 DMA 处理依赖于中断报告。虽然可以使用轮询驱动程序实现 DMA,但这没有任何意义,因为轮询驱动程序会浪费 DMA 提供的性能优势,而不是更容易的处理器驱动的 I/O。*

* 当然,凡事都有例外;请参阅第 17 章中的“接收中断缓解”一节,以演示如何最好地使用轮询来实现高性能网络驱动程序。

这里介绍的另一个相关项目是 DMA 缓冲区。 DMA 要求设备驱动程序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它们的缓冲区并使用它们直到关闭 因此,前面列表中的allocate一词的意思是“获取先前分配的缓冲区”。

分配 DMA 缓冲区本节介绍低级别的

DMA 缓冲区分配;我们很快就会介绍一个更高级别的接口,但是理解这里提供的材料仍然是一个好主意。

DMA 缓冲区出现的主要问题是,当它们大于一页时,它们必须占用物理内存中的连续页,因为设备使用 ISA 或 PCI 系统总线传输数据,两者都携带物理地址。有趣的是,该约束不适用于使用外围总线上的虚拟地址的系统总线(参见第 12 章中的“系统总线”一节)。一些体系结构也可以在 PCI 总线上使用虚拟地址,但便携式驱动程序不能指望这种能力。

尽管 DMA 缓冲区可以在系统启动时或运行时分配,但模块只能在运行时分配它们的缓冲区。 (第 8 章介绍了这些技术;“获取大型缓冲区”部分介绍了系统启动时的分配,而“kmalloc 的真实故事”和“get_free_page 及其朋友”描述了运行时的分配。)驱动程序编写者必须小心分配用于 DMA 操作时的正确类型的内存;并非所有内存区域都适合。特别是,在某些系统和某些设备上,高内存可能不适用于 DMA 外设根本无法使用这么高的地址。

现代总线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它们来说工作得很好。然而,一些 PCI 设备未能实现完整的 PCI 标准,并且无法使用 32 位地址。当然,ISA 设备仅限于 24 位地址。

对于具有这种限制的设备,应通过将GFP_DMA标志添加到kmalloc或get_free_pages调用来从 DMA 区域分配内存。当此标志存在时,仅分配可以用 24 位寻址的内存。或者,您可以使用通用 DMA 层(我们将稍后讨论)来分配缓冲区以解决您设备的限制。

自己动手分配

我们已经看到get_free_pages是如何分配高达几兆字节的(因为 order 的范围可以高达 MAX_ORDER,目前是 11),但是高阶请求甚至容易失败

当请求的缓冲区远小于 128 KB 时,因为系统内存会随着时间的推移而变得碎片化。*当内核无法返回请求的内存量或您需要超过 128 KB 时（例如 PCI 帧捕获器的常见要求），返回-ENOMEM的替代方法是在引导时分配内存或为缓冲区保留物理 RAM 的顶部。我们在第 8 章的“获取大缓冲区”一节中描述了引导时的分配,但它不适用于模块。保留 RAM 的顶部是通过在引导时将mem=参数传递给内核来完成的。例如,如果您有 256 MB,则参数mem=255M会阻止内核使用最高兆字节。您的模块稍后可以使用以下代码来访问此类内存：

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

分配器是本书附带的示例代码的一部分,它提供了一个简单的 API 来探测和管理这些保留的 RAM,并已在多种架构上成功使用。但是,当您拥有一个高内存系统（即物理内存超出 CPU 地址空间的容量的系统)时,此技巧不起作用。

当然,另一种选择是使用GFP_NOFAIL分配标志来分配缓冲区。然而,这种方法确实对内存管理子系统造成了严重的压力,并且存在完全锁定系统的风险;除非真的没有其他办法,否则最好避免。

但是,如果您要花这么多时间来分配一个大的 DMA 缓冲区,那么值得考虑一些替代方案。如果您的设备可以进行分散/收集 I/O,您可以将缓冲区分配为较小的部分,让设备完成其余的工作。在用户空间执行直接 I/O 时也可以使用分散/聚集 I/O,这在需要真正巨大的缓冲区时可能是最佳解决方案。

总线地址

使用 DMA 的设备驱动程序必须与连接到使用物理地址的接口总线的硬件通信,而程序代码使用虚拟地址。

事实上,情况比这要复杂一些。基于 DMA 的硬件使用总线地址,而不是物理地址。尽管 ISA 和 PCI 总线地址只是 PC 上的物理地址,但并非每个平台都如此。有时,接口总线通过将 I/O 地址映射到不同物理地址的桥接电路连接。一些系统甚至有一个页面映射方案,可以使任意页面看起来与外围总线是连续的。

*碎片一词通常用于磁盘,表示文件不是连续存储在磁介质上的想法。同样的概念也适用于内存,其中每个虚拟地址空间分散在整个物理 RAM 中,当请求 DMA 缓冲区时,很难检索连续的空闲页面。

在最低级别（再次,我们将很快看到更高级别的解决方案）,Linux 内核通过导出以下函数提供了一个可移植的解决方案,这些函数在<asm/io.h> 中定义。强烈建议不要使用这些函数,因为它们只能在具有非常简单的 I/O 架构的系统上正常工作;尽管如此,您在使用内核代码时可能会遇到它们。

```
unsigned long virt_to_bus(volatile void
*address); void *bus_to_virt(无符号长地址);这些
```

函数执行内核逻辑地址和总线地址之间的简单转换。它们不适用于必须对 I/O 内存管理单元进行编程或必须使用反弹缓冲区的任何情况。执行此转换的正确方法是使用通用 DMA 层,因此我们现在继续讨论该主题。

通用 DMA 层DMA 操作最终归结

为分配缓冲区并将总线地址传递给您的设备。然而,编写可在所有架构上安全正确地执行 DMA 的可移植驱动程序的任务比人们想象的要难。不同的系统对缓存一致性应该如何工作有不同的想法。如果您没有正确处理此问题,您的驱动程序可能会损坏内存。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执行 DMA。幸运的是,内核提供了一个独立于总线和体系结构的 DMA 层,它对驱动程序作者隐藏了大部分这些问题。我们强烈建议您在您编写的任何驱动程序中将此层用于 DMA 操作。

下面的许多函数都需要一个指向结构设备的指针。此结构是 Linux 设备模型中设备的低级表示。它不是驱动程序经常必须直接使用的东西,但在使用通用 DMA 层时确实需要它。通常,您可以在描述您的设备的特定总线中找到此结构。例如,它可以作为struct pci_device或struct usb_device 中的dev字段找到。第14 章详细介绍了设备结构。

使用以下函数的驱动程序应包括<linux/dma-mapping.h>。

处理困难的硬件在尝试 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执行此类操作。由于多种原因,许多设备在它们可以寻址的内存范围内受到限制。默认情况下,内核假定您的设备可以对任何 32 位地址执行 DMA。如果不是这种情况,您应该通过调用以下命令通知内核该事实：

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

掩码应显示您的设备可以寻址的位;例如,如果它被限制为 24 位,您可以将掩码传递为 0x0FFFFFF。如果给定掩码可以进行 DMA,则返回值非零;如果dma_set_mask返回0,则您无法在此设备上使用 DMA 操作。因此,仅限于 24 位 DMA 操作的设备的驱动程序中的初始化代码可能如下所示:

```
if (dma_set_mask (dev, 0xffffff)) 卡->use_dma =
        1;否则 { 卡->use_dma = 0; printk
(KERN_WARN,    mydev: DMA not supported\n    );
                                        /* 我们将不得不在没有 DMA 的情况下生活 */

}
```

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

DMA 映射

DMA 映射是分配DMA 缓冲区和为该缓冲区生成设备可访问的地址的组合。很容易通过简单调用virt_to_bus 来获取该地址,但有充分的理由避免这种方法。首先是合理的硬件带有一个 IOMMU,它为总线提供一组映射寄存器。 IOMMU 可以安排任何物理内存出现在设备可访问的地址范围内,并且它可以使物理上分散的缓冲区看起来与设备相邻。使用 IOMMU 需要使用通用 DMA 层; virt_to_bus不能胜任这项任务。

请注意,并非所有架构都有 IOMMU。特别是流行的 x86 平台不支持 IOMMU。然而,正确编写的驱动程序不需要知道它所运行的 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建立反弹缓冲区。当驱动程序尝试在外围设备无法访问的地址(例如,高内存地址)上执行 DMA 时,会创建反弹缓冲区。然后根据需要将数据复制到并从反弹缓冲区复制。不用说,使用反弹缓冲区可以减慢速度,但有时别无选择。

DMA 映射还必须解决高速缓存一致性问题。请记住,现代处理器将最近访问的内存区域的副本保存在快速的本地缓存中。没有这个缓存,就不可能有合理的性能。如果您的设备更改了主存储器的某个区域,则必须使覆盖该区域的任何处理器缓存失效;否则处理器可能会处理不正确的主内存映像,并导致数据损坏。类似地,当您的设备使用 DMA 从主内存中读取数据时,必须首先刷新对驻留在处理器缓存中的内存的任何更改。如果程序员不小心,这些缓存一致性问题可能会产生无穷无尽的晦涩难懂和难以发现的错误。一些架构管理缓存

硬件的一致性,但其他需要软件支持。通用 DMA 层竭尽全力确保一切在所有架构上都能正常工作,但是,正如我们将看到的,正确的行为需要遵守一小部分规则。

DMA 映射设置了一个新类型dma_addr_t来表示总线地址。 dma_addr_t类型的变量应该被驱动程序视为不透明的;唯一允许的操作是将它们传递给 DMA 支持例程和设备本身。dma_addr_t作为总线地址,如果直接被 CPU 使用,可能会导致意想不到的问题。

PCI 代码区分两种类型的 DMA 映射,具体取决于 DMA 缓冲区预计保留多长时间： 一致的DMA 映射这些映射通常存在于驱动程序的生命周期中。一个连贯的缓冲区必须同时可供 CPU 和外设使用（其他类型的映射,正如我们稍后将看到的,在任何给定时间只能对其中一个或另一个可用）。因此,一致的映射必须存在于缓存一致的内存中。建立和使用相干映射的成本可能很高。

流式传输 DMA 映射

通常为单个操作设置流式映射。正如我们所看到的,当使用流映射时,一些架构允许进行重大优化,但这些映射在如何访问它们时也受到一组更严格的规则的约束。内核开发人员建议尽可能在连贯映射上使用流映射 ping。这个建议有两个原因。第一个是,在支持映射寄存器的系统上,每个 DMA 映射在总线上使用一个或多个。具有较长生命周期的相干映射可以长期独占这些寄存器,即使它们没有被使用。另一个原因是,在某些硬件上,流映射可以以连贯映射不可用的方式进行优化。

这两种映射类型必须以不同的方式进行操作;是时候看看细节了。

设置一致的 DMA 映射

驱动程序可以通过调用dma_alloc_coherent来设置一致映射：

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

该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缓冲区的大小。该函数在两个地方返回 DMA 映射的结果。函数的返回值是缓冲区的内核虚拟地址,驱动程序可以使用该地址;同时,相关的总线地址在dma_handle 中返回。分配处理在

此函数使缓冲区放置在与 DMA 一起使用的位置;通常内存只是用get_free_pages分配的（但请注意,大小以字节为单位,而不是顺序值）。 flag参数是描述如何分配内存的常用GFP_值;它通常应该是GFP_KERNEL （通常)或GFP_ATOMIC （在原子上下文中运行时）。

当不再需要缓冲区时（通常在模块卸载时）,应使用dma_free_coherent 将其返回给系统：

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

请注意,此函数与许多通用 DMA 函数一样,需要提供所有大小、CPU 地址和总线地址参数。

DMA 池

DMA 池是一种用于小型、一致的 DMA 映射的分配机制。从dma_alloc_coherent获得的映射 ping的最小大小可能为一页。如果您的设备需要比这更小的 DMA 区域,您可能应该使用 DMA 池。DMA 池在您可能想要对嵌入在较大结构中的小区域执行 DMA 的情况下也很有用。一些非常模糊的驱动程序错误已被追溯到与小 DMA 区域相邻的结构字段的缓存一致性问题。为避免此问题,您应始终明确地为 DMA 操作分配区域,远离其他非 DMA 数据结构。

DMA 池函数在<linux/dmapool.h> 中定义。

使用前必须创建 DMA 池,并调用：

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

这里， name是池的名称， dev是您的设备结构， size是要从该池中分配的缓冲区的大小， align是从池中分配所需的硬件对齐（以字节表示）,分配是,如果非零,则分配不应超过的内存边界。例如,如果分配作为 4096 传递,则从此池分配的缓冲区不会跨越 4 KB 边界。

完成池后,可以通过以下方式释放它：

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

您应该在销毁池之前将所有分配归还给池。

分配由dma_pool_alloc 处理： void

```
*dma_pool_alloc(struct dma_pool *pool, int mem_flags,
                dma_addr_t *句
```

柄）;对于这个调用， mem_flags是一组常用的GFP_分配标志。如果一切顺利,将分配一个内存区域（在创建池时指定大小)并

回来。与dma_alloc_coherent 一样,生成的 DMA 缓冲区的地址作为内核虚拟地址返回,并作为总线地址存储在句柄中。

不需要的缓冲区应该返回到池中:

```
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,流式

映射比连贯的映射具有更复杂的接口。这些映射期望使用驱动程序已经分配的缓冲区,因此必须处理他们没有选择的地址。在某些架构上,流映射还可以有多个不连续的页面和多部分“分散/收集”缓冲区。由于所有这些原因,流映射有自己的一组映射函数。

设置流映射时,您必须告诉内核数据移动的方向。为此目的定义了一些符号(类型为enum dma_data_direction):

DMA_TO_DEVICE
: DMA_FROM_DEVICE这两个符号应该是不言自明的。如果正在向设备发送数据(可能是为了响应写入系统调用),则应使用DMA_TO_DEVICE ;相反,进入 CPU 的数据用 DMA_FROM_DEVICE 标记。

DMA_BIDIRECTIONAL
: 如果数据可以在任一方向移动,请使用DMA_BIDIRECTIONAL。

DMA_NONE此符号仅作为调试帮助提供。尝试使用这个“方向”的缓冲区会导致内核恐慌。

始终选择DMA_BIDIRECTIONAL可能很诱人,但驱动程序作者应该抵制这种诱惑。在某些架构上,这种选择会带来性能损失。

当你有一个缓冲区要传输时,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);
```

返回值是您可以传递给设备的总线地址,如果出现问题,则返回NULL 。

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 方向);在这
```

里,大小和方向参数必须与用于映射缓冲区的参数相匹配。

一些重要的规则适用于流式 DMA 映射：

- 缓冲区必须仅用于与方向值匹配的传输映射时给出。
- 一旦缓冲区被映射，它就属于设备，而不是处理器。在缓冲区未映射之前，驱动程序不应以任何方式接触其内容。只有在调用 dma_unmap_single之后，驱动程序才能安全地访问缓冲区的内容（我们很快就会看到一个例外）。

  除其他外，此规则意味着写入设备的缓冲区在包含所有要写入的数据之前不能被映射。
- 当 DMA 仍处于活动状态时，不得取消映射缓冲区，否则会导致严重的系统不稳定。

您可能想知道为什么一旦缓冲区被映射，驱动程序就不能再使用它了。这条规则之所以有意义，实际上有两个原因。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都已实际写入内存。发出dma_unmap_single时，处理器的缓存中可能有一些数据，必须显式刷新。刷新后处理器向缓冲区写入 10 的数据可能对设备不可见。

其次，考虑如果要映射的缓冲区位于设备无法访问的内存区域中会发生什么。在这种情况下，一些架构会简单地失败，但其他架构会创建反弹缓冲区。反弹缓冲区只是设备可访问的单独内存区域。如果缓冲区以DMA_TO_DEVICE方向映射，并且需要反弹缓冲区，则原始缓冲区的内容将作为映射操作的一部分进行复制。显然，设备看不到复制后对原始缓冲区的更改。类似地， DMA_FROM_DEVICE反弹缓冲区由dma_unmap_single 复制回原始缓冲区；在复制完成之前，设备中的数据不存在。

顺便说一句，反弹缓冲区是正确选择方向很重要的原因之一。 DMA_BIDIRECTIONAL反弹缓冲区在操作之前和之后都被复制，这通常是不必要的 CPU 周期浪费。

有时，驱动程序需要访问流式 DMA 缓冲区的内容而不取消映射。已经提供了一个电话来使这成为可能：

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

应该在处理器访问流式 DMA 缓冲区之前调用此函数。一旦调用完成，CPU “拥有” DMA 缓冲区并可以根据需要使用它。然而，在设备访问缓冲区之前，所有权应该通过以下方式转移回它：

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

再次,处理器不应在此调用后访问 DMA 缓冲区。

单页面流映射例如,这可能发生在使

用get_user_pages 映射的用户空间缓冲区。要使用结构页面指针设置和拆除流映射,请使用以下命令:

```
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 方向);
```

偏移量和大小参数可用于映射页面的一部分。但是,建议避免部分页面映射,除非您真的确定自己在做什么。如果分配仅覆盖缓存行的一部分,则映射页面的一部分可能会导致缓存一致性问题;这反过来又会导致内存损坏和极难调试的错误。

Scatter/gather 映射

Scatter/gather 映射是一种特殊类型的流式 DMA 映射。假设您有多个缓冲区,所有这些缓冲区都需要传入或传出设备。这种情况可能以多种方式出现,包括readv或writev系统调用、集群磁盘 I/O 请求或映射内核 I/O 缓冲区中的页面列表。

您可以简单地依次映射每个缓冲区并执行所需的操作,但是一次映射整个列表有好处。

许多设备可以接受数组指针和长度的分散列表,并在一次 DMA 操作中传输它们;例如,如果可以将数据包构建成多个部分,那么“零拷贝”网络就更容易了。将分散列表作为一个整体进行映射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,物理上不连续的页面可以组装成单个连续的阵列。此技术仅在分散列表中的条目长度等于页面大小时才有效(第一个和最后一个除外),但当它确实有效时,它可以将多个操作转换为单个 DMA,并相应地加快处理速度。

最后,如果必须使用反弹缓冲区,则将整个列表合并到一个缓冲区中是有意义的(因为它无论如何都会被复制)。

所以现在您确信散点表的映射在某些情况下是值得的。映射 scatterlist 的第一步是创建并填充 struct scatterlist数组,该数组描述要传输的缓冲区。这种结构是建筑

依赖,并在<asm/scatterlist.h> 中描述。但是,它始终包含三个字段:

结构页*页;与要在分
散/收集操作中使用的缓冲区相对应的结构页指针。无符号整数长度;无符号整数偏移量;该缓冲区的长度及其在页面内的偏移量

要映射分散/聚集 DMA 操作,您的驱动程序应在struct scatterlist条目中为要传输的每个缓冲区设置页面、偏移量和长度字段。然后调用:

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

direction)其中nents是传入的 scatterlist 条目数。返回值是要传输的 DMA 缓冲区数;它可能低于nts。

对于输入分散列表中的每个缓冲区， dma_map_sg确定要提供给设备的正确总线地址。作为该任务的一部分,它还会合并内存中彼此相邻的缓冲区。如果您的驱动程序正在运行的系统有一个 I/O 内存管理单元， dma_map_sg也会对该单元的映射寄存器进行编程,从您的设备的角度来看,您可以传输单个连续的缓冲区。但是,在通话结束之前,您永远不会知道最终的转移会是什么样子。

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,但它们在结构中的位置因架构而异。定义了两个宏以使编写可移植代码成为可能： dma_addr_t sg_dma_address(struct scatterlist *sg);

从该 scatterlist 条目返回总线 (DMA) 地址。

无符号整数 sg_dma_len(struct scatterlist *sg);返回此
缓冲区的长度。

再次记住,要传输的缓冲区的地址和长度可能与传入dma_map_sg 的不同。

传输完成后,通过调用dma_unmap_sg 取消映射分散/聚集映射:

```
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,而不是函数返回给您的 DMA 缓冲区数。

Scatter/gather 映射是流式 DMA 映射,对它们适用与单一品种相同的访问规则。如果您必须访问映射的分散/聚集列表,则必须首先对其进行同步:

```
void dma_sync_sg_for_cpu（结构设备 *dev,结构分散列表 *sg,
                          int nents,枚举 dma_data_direction 方向）； void
dma_sync_sg_for_device(struct device *dev, struct scatterlist *sg, int nents, enum
                          dma_data_direction direction);
```

PCI 双地址周期映射通常,DMA 支持

层使用 32 位总线地址,可能受特定设备的 DMA 掩码限制。然而,PCI 总线也支持 64 位寻址模式,即双地址周期(DAC)。通用 DMA 层不支持这种模式有几个原因,首先是它是 PCI 特定的功能。此外,DAC 的许多实现充其量都是错误的,而且由于 DAC 比常规的 32 位 DMA 慢,因此可能会产生性能成本。

即便如此,在某些应用程序中使用 DAC 可能是正确的做法。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,您可能需要考虑实现 DAC 支持。此支持仅适用于 PCI 总线,因此必须使用特定于 PCI 的例程。

要使用 DAC,您的驱动程序必须包含<linux/pci.h>。您必须设置单独的 DMA 掩码:

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

仅当此调用返回0时,您才能使用 DAC 寻址。

一个特殊类型(dma64_addr_t)用于 DAC 映射。要建立这些映射之一,请调用 pci_dac_page_to_dma:

```
dma64_addr_t pci_dac_page_to_dma(struct pci_dev *pdev, struct page *page,
                                 无符号长偏移量,整数方向）；
```

您会注意到,DAC 映射只能由结构页指针生成（毕竟它们应该存在于高内存中,否则使用它们毫无意义）;它们必须一次创建一个页面。方向参数是通用 DMA 层中使用的枚举 dma_data_direction 的PCI 等效项;它应该是PCI_DMA_TODEVICE、 PCI_DMA_FROMDEVICE或 PCI_DMA_BIDIRECTIONAL。

DAC 映射不需要外部资源,因此无需在使用后显式释放它们。但是,有必要像对待其他流映射一样对待 DAC 映射,并遵守有关缓冲区所有权的规则。有一组与通用类型类似的用于同步 DMA 缓冲区的函数:

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

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

一个简单的 PCI DMA 示

例作为如何使用 DMA 映射的示例,我们提供了一个用于 PCI 设备的 DMA 编码的简单示例。 PCI 总线上 DMA 操作的实际形式非常依赖于被驱动的设备。因此,此示例不适用于任何真实设备;相反,它是名为dad （DMA 采集设备)的假设驱动程序的一部分。此设备的驱动程序可能会定义如下传输函数：

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

    /* 为 DMA 映射缓冲区 */ dev-
    >dma_dir = (write ? DMA_TO_DEVICE : DMA_FROM_DEVICE); dev-
    >dma_size = 计数； bus_addr = dma_map_single(&dev->pci_dev-
    >dev,缓冲区,计数,dev->dma_dir);开发->dma_addr = bus_addr；



    /* 设置设备 */

    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));

    /* 开始操作 */ writeb(dev-
    >registers.command, DAD_CMD_ENABLEDMA);返回0；

}
```

此函数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,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;

    /* 确保它真的是我们的设备中断 */

    /* 取消映射 DMA 缓冲区 */
    dma_unmap_single(dev->pci_dev->dev, dev->dma_addr,
                     dev->dma_size, dev->dma_dir);

    /* 只有现在才可以安全地访问缓冲区、复制给用户等 */
    ...
}
```

显然,该示例遗漏了大量细节,包括可能需要哪些步骤来防止尝试启动多个同时的 DMA 操作。

用于 ISA 设备的 DMA

ISA 总线允许两种 DMA 传输:本地 DMA 和 ISA 总线主控 DMA。本机 DMA 使用主板上的标准 DMA 控制器电路来驱动 ISA 总线上的信号线。另一方面,ISA 总线主控 DMA 完全由外围设备处理。后一种类型的 DMA 很少使用,这里不需要讨论,因为它类似于 PCI 设备的 DMA,至少从驱动程序的角度来看是这样。 ISA 总线主控器的一个示例是 1542 SCSI 控制器,其驱动程序是内核源代码中的drivers/scsi/aha1542.c 。

就本机 DMA 而言,ISA 总线上的 DMA 数据传输涉及三个实体:

8237 DMA 控制器 (DMAC)
: 控制器保存有关 DMA 传输的信息,例如方向、内存地址和传输大小。它还包含一个跟踪正在进行的传输状态的计数器。当控制器接收到 DMA 请求信号时,它会控制总线并驱动信号线,以便设备可以读取或写入其数据。

外围设备当设备准备好
: 传输数据时,它必须激活 DMA 请求信号。实际传输由 DMAC 管理;当控制器选通设备时,硬件设备按顺序读取或写入数据到总线上。

: 传输结束时,设备通常会引发中断。

设备驱动程序与驱
: 动程序无关;它为 DMA 控制器提供传输的方向、总线地址和大小。它还与外围设备对话以准备传输数据,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”,每个通道与一组 DMA 寄存器相关联。四个设备可以同时将它们的 DMA 信息存储在控制器中。较新的 PC 包含相当于两个 DMAC 设备:*第二个控制器(主)连接到系统处理器,第一个(从)连接到第二个控制器的通道0。 †

* 这些电路现在是主板芯片组的一部分,但几年前它们是两个独立的 8237 芯片。

† 最初的 PC 只有一个控制器;第二个是在基于 286 的平台中添加的。但是,第二个控制器作为主控制器连接,因为它处理 16 位传输;第一个一次只传输八位,并且是为了向后兼容。

通道编号从 0 到 7:通道 4 对 ISA 外设不可用，
因为它在内部用于将从控制器级联到主控制器。这
因此,可用通道在从机（8 位通道)上为 0-3,在从机上为 5-7
主控（16 位通道）。存储在控制器中的任何 DMA 传输的大小是一个 16 位数字,表示总线周期数。因此,从控制器的最大传输大小为 64 KB（因为它传输 8 位

一个周期)和 128 KB 用于主机（进行 16 位传输）。

因为 DMA 控制器是系统范围的资源,所以内核会帮助处理它。
它使用 DMA 注册表为 DMA 通道提供请求和释放机制,并使用一组函数在 DMA 控制器中配置通道信息。

注册 DMA 使用

您应该习惯于内核注册表 我们已经看到它们用于 I/O 端口和
中断线。 DMA 通道注册与其他类似。在<asm/dma.h>之后
已包含以下功能,可用于获取和释放 DMA 通道的所有权：

```
int request_dma(unsigned int channel, const char *name);
无效free_dma（无符号整数通道）；
```

通道参数是一个介于 0 和 7 之间的数字,或者更准确地说,是一个正数
数量小于MAX_DMA_CHANNELS。在 PC 上， MAX_DMA_CHANNELS定义为8到
匹配硬件。 name参数是一个标识设备的字符串。指定的名称出现在文件/proc/dma 中,用户程序可以读取该文件。

request_dma的返回值为0表示成功， -EINVAL或-EBUSY表示成功
一个错误。前者表示请求的频道超出范围,后者
表示另一台设备正在持有该频道。

我们建议您对 DMA 通道和 I/O 端口一样小心
和中断线路;在打开时请求通道比从模块初始化函数中请求通道要好得多。延迟请求允许驱动程序之间进行一些共享;例如,您的声卡和模拟 I/O 接口可以

只要不同时使用,就共享 DMA 通道。

我们还建议您在请求中断线后请求 DMA 通道,并在中断前释放它。这是常规顺序

用于请求这两个资源;遵循约定可以避免可能的死锁。请注意,每个使用 DMA 的设备也需要一条 IRQ 线;否则,它
无法发出数据传输完成的信号。

在典型情况下， open的代码如下所示,它引用了我们假设的dad模块。如图所示的爸爸设备使用快速中断处理程序,没有
支持共享 IRQ 线路。

```
int dad_open (struct inode *inode, struct file *filp)
{
    结构爸爸设备*我的设备；
```

```
        /* ... */ if
        ( (error = request_irq(my_device.irq, dad_interrupt, SA_INTERRUPT,    dad    ,
                                                    NULL)) ) 返回错误；  /* 或者实现阻塞打开 */


        if ( (error = request_dma(my_device.dma,    dad    )) )
              { free_irq(my_device.irq, NULL);返回错误；  /* 或者实现阻塞打开 */


        } /* ... */
        返回0；
}
```

与刚刚显示的open匹配的close实现如下所示： void dad_close (struct inode *inode, struct file *filp) {

```
        结构爸爸设备*我的设备；

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

}
```

以下是/proc/dma文件在安装了声卡的系统上的外观：

```
merlino%猫 /proc/dma
  1:Sound Blaster8 4:级联
```

有趣的是,默认声音驱动程序在系统启动时获取 DMA 通道并且从不释放它。级联条目是一个占位符,表示通道 4对驱动程序不可用,如前所述。

与 DMA 控制器对话 注册后，

驱动程序工作的主要部分包括配置 DMA 控制器以使其正常运行。这项任务并非微不足道,但幸运的是,内核导出了典型驱动程序所需的所有功能。

驱动程序需要在调用read或write或准备异步传输时配置 DMA 控制器。后一项任务要么在打开时执行,要么响应ioctl命令执行,具体取决于驱动程序和它实现的策略。此处显示的代码是通常由读取或写入设备方法调用的代码。

本小节简要介绍了 DMA 控制器的内部结构,以便您了解此处介绍的代码。如果您想了解更多信息,我们建议您阅读<asm/dma.h>和一些描述 PC 架构的硬件手册。在

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，您应该在设备的硬件手册中找到相关信息。

DMA 控制器是一种共享资源,如果多个处理器尝试同时对其进行编程,可能会出现混乱。出于这个原因,控制器受到一个自旋锁的保护,称为dma_spin_lock。司机不应直接操作锁;但是,已经为您提供了两个功能：

unsigned long claim_dma_lock( );
: 获取 DMA 自旋锁。此函数还阻止本地处理器上的中断;因此,返回值是一组描述先前中断状态的标志;完成锁定后,必须将其传递给以下函数以恢复中断状态。

void release_dma_lock（无符号长标志）;返回
: DMA 自旋锁并恢复之前的中断状态。

使用下面描述的功能时应该保持自旋锁。但是,它不应该在实际 I/O 期间保持。持有自旋锁时,驱动程序不应该睡觉。

必须加载到控制器中的信息包括三项:RAM 地址、必须传输的原子项数（以字节或字为单位)以及传输方向。为此， <asm/dma.h> 导出了以下函数：

void set_dma_mode(unsigned int channel, char mode);
: 指示通道是必须从设备读取(DMA_MODE_READ)还是写入设备 (DMA_MODE_WRITE)。存在第三种模式， DMA_MODE_CASCADE,用于释放对总线的控制。级联是第一个控制器连接到第二个控制器顶部的方式,但它也可以被真正的 ISA 总线主控设备使用。我们不会在这里讨论总线控制。 void set_dma_addr（无符号整数通道,无符号整数地址）;分配 DMA 缓冲区的地址。该函数将addr的 24 个最低有效位存储在控制器中。 addr参数必须是总线地址（参见本章前面的“总线地址”部分）。 void set_dma_count（无符号整数通道,无符号整数计数）;分配要传输的字节数。 count参数也表示 16 位通道的字节数;在这种情况下,数字必须是偶数。

除了这些函数之外,在处理 DMA 设备时还必须使用许多内务处理工具： void
disable_dma(unsigned int channel);

可以在控制器内禁用 DMA 通道。在配置控制器之前应禁用该通道,以防止不当操作。 （否则，可能会发生损坏,因为控制器是通过 8 位数据传输进行编程的,因此之前的功能都不是原子执行的）。

无效 enable_dma（无符号整数通道）;该函
数告诉控制器 DMA 通道包含有效数据。

int get_dma_residue（无符号整数通道）;驱动
程序有时需要知道 DMA 传输是否已完成。此函数返回仍要传输的字节数。

成功传输后返回值为0 ,并且在控制器工作时是不可预测的（但不是0）。不可预测性源于需要通过两个 8 位输入操作获得 16 位余数。

void clear_dma_ff（无符号整数通道）

该函数清除 DMA 触发器。触发器用于控制对 16 位寄存器的访问。寄存器通过两个连续的 8 位操作访问,触发器用于选择最低有效字节（当它被清除时)或最高有效字节（当它被设置时）。当传输了八位时,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）。

使用这些函数,驱动程序可以实现如下函数来为 DMA 传输做准备：

```
int dad_dma_prepare(int channel, int mode, unsigned int buf, unsigned
                                int count)
{
    无符号长标志;

    标志 = claim_dma_lock( );
    disable_dma（频道）;
    clear_dma_ff（频道）;
    set_dma_mode（通道,模式）;
    set_dma_addr（通道,virt_to_bus（buf））;
    set_dma_count（通道,计数）; enable_dma
    （频道）; release_dma_lock（标志）;


    返回0;
}
```

然后,类似下一个的函数用于检查是否成功完成
DMA：

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

```
        残基;无符号长
        标志 = claim_dma_lock ( );残留物 =
        get_dma_residue（通道）； release_dma_lock
        （标志）;返回（残基 = = 0）；

    }
```

唯一需要做的就是配置设备板。此设备特定任务通常包括读取或写入一些 I/O 端口。设备有很大的不同。例如，一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线到设备中的值。对于配置板，硬件手册是您唯一的朋友。

快速参考

本章介绍了以下与内存处理相关的符号。

介绍材料

#include <linux/mm.h>

#include <asm/page.h>

大多数与内存管理相关的函数和结构都是原型类型的，并在这些头文件中定义。

void *__va(unsigned long physaddr);无符号长

__pa(void *kaddr);

在内核逻辑地址和物理地址之间转换的宏。

PAGE_SIZE

PAGE_SHIFT

给出底层硬件上页面大小（以字节为单位）的常量，以及为将页面帧号转换为物理地址而必须移动的位数。

结构页面

表示系统内存映射中的硬件页面的结构。结构页 *virt_to_page(void *kaddr);

无效*page_address（结构页面*page）；结构页面 *pfn_to_page(int pfn);

在内核逻辑地址及其关联的内存映射条目之间进行转换的宏。 page_address仅适用于已显式映射的低内存页面或高内存页面。 pfn_to_page将页帧号转换为其关联的结构页指针。

无符号长kmap（结构页*页）;无效kunmap（结构页*
页）； kmap返回一个映射到给定页面的内核虚拟地
址,如果需要,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;生成的映射只能由原子代码保存。对于驱动程序,类型应为KM_USER0、KM_USER1、 KM_IRQ0或KM_IRQ1。结构 vm_area_struct;描述 VMA 的结构。

实现 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核心的函数。它们映射大小字节的物理地址,从 pfn 指示的页号开始到虚拟地址virt_add 。与虚拟空间相关的保护位在 prot 中指定。 io_remap_page_range应该在目标地址在 I/O 时使用

内存空间。

结构页 *vmalloc_to_page(void *vmaddr);将从vmalloc获得的内核虚拟地址转换为其对应的结构页指针。

实现直接 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并返回相应结构页指针的函数。调用者必须持有mm->mmap_sem。

SetPageDirty(struct page *page);将给定页面标记为“脏”（已修改)并且需要在释放之前写入其后备存储的宏。 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);
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);
指示异步 I/O 操作完成的函数。

直接内存访问

#include <asm/io.h>
unsigned long virt_to_bus(volatile void * address); void *
bus_to_virt(无符号长地址);在内核、虚拟和总线地址之间转换的
过时和不推荐使用的函数。必须使用总线地址与外围设备通信。

#include <linux/dma-mapping.h>
定义通用 DMA 函数所需的头文件。
int dma_set_mask(struct device *dev, u64 mask);对于
无法寻址完整 32 位范围的外设,此函数将可寻址范围通知内核,如果可以使用 DMA,则返回非零
值。
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 总线地址）；
为将持续驱动程序生命周期的缓冲区分配和释放一致的 DMA 映射。

#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);
无效 dma_pool_destroy(struct dma_pool *pool);
void *dma_pool_alloc（结构 dma_pool *pool,int mem_flags,dma_addr_t
*处理）；
void dma_pool_free(struct dma_pool *pool, void *vaddr, dma_addr_t 句柄);
创建、销毁和使用 DMA 池来管理小型 DMA 区域的函数。
枚举 dma_data_direction；
DMA_TO_DEVICE
DMA_FROM_DEVICE
DMA_BIDIRECTIONAL
DMA_NONE用于告诉流映射
函数数据移入或移出缓冲区的方向的符号。

dma_addr_t dma_map_single(struct device *dev, void *buffer, size_t size, enum
dma_data_direction 方向）；
void dma_unmap_single(struct device *dev, dma_addr_t bus_addr, size_t size, enum dma_data_direction
方向);
创建和销毁一次性的流式 DMA 映射。 void dma_sync_single_for_cpu（结构设
备 *dev,dma_handle_t bus_addr,size_t
大小,枚举 dma_data_direction 方向）；
void dma_sync_single_for_device(struct device *dev, dma_handle_t bus_addr, size_t size, enum
dma_data_direction 方向);
同步具有流映射的缓冲区。如果处理器必须在流映射到位时（即设备拥有缓冲区)访问缓冲区，
则必须使用这些函数。

#include <asm/scatterlist.h>
struct scatterlist { /* ... */ };
dma_addr_t sg_dma_address(struct scatterlist *sg);无
符号整数 sg_dma_len(struct scatterlist *sg); scatterlist
结构描述了一种涉及多个缓冲区的 I/O 操作。宏sg_dma_address和sg_dma_len可用于在
实现分散/聚集操作时提取总线地址和缓冲区长度以传递给设备。

dma_map_sg(struct device *dev, struct scatterlist *list, int nents,
枚举 dma_data_direction 方向）；
dma_unmap_sg(struct device *dev, struct scatterlist *list, int nents, enum
dma_data_direction 方向）； void
dma_sync_sg_for_cpu（结构设备 *dev,结构 scatterlist *sg,int
nents,枚举 dma_data_direction 方向）；
无效 dma_sync_sg_for_device（结构设备 *dev,结构分散列表 *sg,int
nents,枚举 dma_data_direction 方向）；
dma_map_sg映射分散/聚集操作,并且dma_unmap_sg撤消该映射。如果必须在映射处于
活动状态时访问缓冲区,则可以使用dma_sync_sg_*来同步事物。 /proc/dma包含 DMA
控制器中分配通道的文本快照的文件。未显示基于 PCI 的 DMA,因为每个板都独立工作,无需
在 DMA 控制器中分配通道。

#include <asm/dma.h>
定义或原型化与 DMA 相关的所有函数和宏的头文件。
必须包含它才能使用以下任何符号。

int request_dma(unsigned int channel, const char *name);无效
free_dma（无符号整数通道）；
访问 DMA 注册表。必须在使用 ISA DMA 通道之前进行注册。

unsigned long claim_dma_lock( );
void release_dma_lock（无符号长标志）；获取和释
放 DMA 自旋锁，必须在调用此列表后面描述的其他 ISA DMA 函数之前保持该自旋锁。它们还禁用和重新启用本地处理器上的中断。 void set_dma_mode(unsigned int channel, char mode);
void set_dma_addr（无符号整数通道，无符号整数地址）； void set_dma_count（无符号整数通道，无符号整数计数）；

在 DMA 控制器中编程 DMA 信息。 addr是总线地址。
void disable_dma（无符号整数通道）；无效
enable_dma（无符号整数通道）；
在配置期间必须禁用 DMA 通道。这些函数改变 DMA 通道的状态。

int get_dma_residue（无符号整数通道）；如果
驱动程序需要知道 DMA 传输是如何进行的，它可以调用此函数，该函数返回尚未完成的数据传输数。 DMA成功完成后，函数返回0；在传输数据时，该值是不可预测的。

void clear_dma_ff（无符号整数通道）
控制器使用 DMA 触发器通过两个 8 位操作传输 16 位值。在向控制器发送任何数据之前必须清除它。