第八章 分配内存

到目前为止，我们已经使用 kmalloc 和 kfree 来分配和释放内存。然而，Linux 内核提供了一套更丰富的内存分配原语。在这一章中，我们将探讨设备驱动程序中使用内存的其他方式，以及如何优化系统的内存资源。我们不会深入到不同架构如何实际管理内存的问题。模块不涉及分段、分页等问题，因为内核为驱动程序提供了统一的内存管理接口。此外，我们在这一章中也不会描述内存管理的内部细节，而是推迟到第15章。

8.1. kmalloc 的真实故事

kmalloc 分配引擎是一个强大的工具，由于其与 malloc 的相似性，易于学习。该函数速度快（除非它阻塞）并且不会清除它获取的内存；分配的区域仍然保持其之前的内容。分配的区域在物理内存中也是连续的。在接下来的几节中，我们将详细讨论 kmalloc，以便你可以将其与我们稍后讨论的内存分配技术进行比较。

8.1.1. flags 参数

```c
#include <linux/slab.h>

void *kmalloc(size_t size, gfp_t flags);
```

- kmalloc 函数接受两个参数：要分配的字节数和分配的标志。返回的是一个指向分配内存的指针，如果分配失败则返回 NULL。
- 标志参数是一个位掩码，用于控制分配的行为。最常用的标志是 GFP_KERNEL，它表示常规的内核内存分配。其他可用的标志包括 GFP_ATOMIC（在不能睡眠的情况下使用，如中断处理程序中）、GFP_DMA（用于满足某些设备的 DMA 限制）等。
- 需要注意的是，kmalloc 分配的内存区域在物理内存中是连续的，但在虚拟内存中可能不连续。如果需要在虚拟内存中也连续的内存区域，应使用 vmalloc 函数。

kmalloc 的第一个参数是要分配的块的大小。第二个参数，分配标志，更有趣，因为它以多种方式控制 kmalloc 的行为。

最常用的标志 GFP*KERNEL，意味着分配（最终通过调用 `__get_free_pages` 内部执行，这是 `GFP` 前缀的来源）*是代表在内核空间运行的进程执行的。换句话说，这意味着调用函数正在代表一个进程执行系统调用。使用 GFP_KERNEL 意味着 kmalloc 可以在低内存情况下调用时将当前进程置于等待页面的状态。因此，使用 GFP_KERNEL 分配内存的函数必须是可重入的，并且不能在原子上下文中运行。当当前进程睡眠时，内核采取适当的行动来找到一些空闲内存，要么通过将缓冲区刷新到磁盘，要么通过从用户进程中交换出内存。

GFP_KERNEL 并不总是正确的分配标志；有时 kmalloc 是从进程上下文之外调用的。例如，这种类型的调用可以发生在中断处理程序、tasklets 和内核定时器中。在这种情况下，不应该让当前进程进入睡眠状态，驱动程序应该使用 GFP_ATOMIC 标志。内核通常会尝试保留一些空闲页面以满足原子分配。当使用 GFP_ATOMIC 时，kmalloc 甚至可以使用最后一个空闲页面。然而，如果最后一个页面不存在，分配就会失败。

除了 `GFP_KERNEL` 和 `GFP_ATOMIC`，还可以使用其他标志，尽管这两个标志覆盖了设备驱动程序的大部分需求。所有的标志都在 <linux/gfp.h> 中定义，每个标志都以双下划线为前缀，如 `__GFP_DMA`。此外，还有一些符号代表常用的标志组合；这些符号没有前缀，有时被称为分配优先级。后者包括：

- GFP_ATOMIC：用于从中断处理程序和进程上下文之外的其他代码分配内存。永不睡眠。
- GFP_KERNEL：正常分配内核内存。可能会睡眠。
- GFP_USER：用于为用户空间页面分配内存；可能会睡眠。
- GFP_HIGHUSER：与 GFP_USER 相似，但如果有的话，会从高内存分配。高内存在下一小节中描述。
- GFP_NOIO
- GFP_NOFS：这些标志的功能类似于 GFP_KERNEL，但它们对内核满足请求可以做什么增加了限制。GFP_NOFS 分配不允许执行任何文件系统调用，而 GFP_NOIO 不允许启动任何 I/O。它们主要用于文件系统和虚拟内存代码，其中分配可能被允许睡眠，但递归文件系统调用将是一个坏主意。

上面列出的分配标志可以通过 ORing 任何以下标志来增强，这些标志改变了分配的执行方式：

- `__GFP_DMA`：此标志请求分配发生在 DMA-capable 内存区域。具体含义取决于平台，并在下一节中解释。
- `__GFP_HIGHMEM`：此标志表示分配的内存可能位于高内存中。
- `__GFP_COLD`：通常，内存分配器试图返回“缓存热”页面——可能在处理器缓存中找到的页面。相反，此标志请求一个“冷”页面，这个页面在一段时间内没有被使

用。它对于分配 DMA 读取的页面很有用，其中处理器缓存的存在并不有用。请参阅第15章的“直接内存访问”部分，以全面讨论如何分配 DMA 缓冲区。

- `__GFP_NOWARN`：这个很少使用的标志阻止内核在无法满足分配时发出警告（使用 printk）。
- `__GFP_HIGH`：此标志标记一个高优先级请求，它被允许消耗内核为紧急情况预留的最后几页内存。
- `__GFP_REPEAT`
- `__GFP_NOFAIL`
- `__GFP_NORETRY`：这些标志修改了分配器在满足分配有困难时的行为。`__GFP_REPEAT` 意味着“尝试一点点更努力”通过重复尝试——但分配仍然可能失败。`__GFP_NOFAIL` 标志告诉分配器永远不要失败；它尽可能地满足请求。强烈不建议使用 `__GFP_NOFAIL`；在设备驱动程序中可能永远没有有效的理由使用它。最后，`__GFP_NORETRY` 告诉分配器如果请求的内存不可用，立即放弃。

8.1.1.1. 内存区

`__GFP_DMA` 和 `__GFP_HIGHMEM` 都有一个取决于平台的角色，尽管它们对所有平台都是有效的。

Linux 内核知道至少三个内存区域：DMA-capable 内存、正常内存和高内存。虽然分配通常发生在正常区域，但设置上述任何一位都需要从不同的区域分配内存。这个想法是，每个必须知道特殊内存范围的计算机平台（而不是考虑所有 RAM 等价物）都将落入这个抽象。

DMA-capable 内存是位于优先地址范围的内存，外设可以在这里执行 DMA 访问。在大多数理智的平台上，所有的内存都在这个区域。在 x86 上，DMA 区域用于前 16 MB 的 RAM，其中旧的 ISA 设备可以执行 DMA；PCI 设备没有这样的限制。

高内存是一种用于在 32 位平台上访问（相对）大量内存的机制。这种内存不能在没有首先设置特殊映射的情况下直接从内核访问，通常更难处理。然而，如果你的驱动程序使用大量的内存，它在大系统上会工作得更好，如果它可以使用高内存。请参阅第15章的“高和低内存”部分，了解高内存的工作方式以及如何使用它。

每当为满足内存分配请求分配新页面时，内核构建一个可以在搜索中使用的区域列表。如果指定了 `__GFP_DMA`，只搜索 DMA 区域：如果在低地址没有可用的内存，分配失败。如果没有特殊的标志，正常和 DMA 内存都会被搜索；如果设置了 `__GFP_HIGHMEM`，所有三个区域都会被用来搜索空闲页面。（然而，请注意，kmalloc 不能分配高内存。）

在非均匀内存访问（NUMA）系统上，情况更复杂。作为一般规则，分配器试图找到本地处理器执行分配的内存，尽管有改变这种行为的方法。

内存区域背后的机制在 mm/page_alloc.c 中实现，而区域的初始化位于平台特定的文件中，通常在 arch 树的 mm/init.c 中。我们将在第15章中重新讨论这些主题。

8.1.2. size 参数

内核管理系统的物理内存，这些内存只以页面大小的块可用。因此，kmalloc 与典型的用户空间 malloc 实现看起来相当不同。一个简单的，基于堆的分配技术会很快遇到麻烦；它在处理页面边界时会有困难。因此，内核使用一种特殊的面向页面的分配技术，以最大限度地利用系统的 RAM。

Linux 通过创建一组固定大小的内存对象池来处理内存分配。分配请求通过去到一个持有足够大对象的池，并将整个内存块返回给请求者来处理。内存管理方案相当复杂，对设备驱动程序编写者来说，它的细节通常并不那么有趣。

然而，驱动程序开发者应该记住的一件事是，内核只能分配某些预定义的，固定大小的字节数组。如果你请求任意数量的内存，你可能会得到 slightly more than you asked for，最多两倍。此外，程序员应该记住，kmalloc 可以处理的最小分配是 32 或 64 字节，这取决于系统架构使用的页面大小。

kmalloc 可以分配的内存块的大小有一个上限。这个限制取决于架构和内核配置选项。如果你的代码要完全可移植，它不能指望能够分配任何大于 128 KB 的东西。然而，如果你需要超过几千字节，那么有比 kmalloc 更好的方法来获取内存，我们将在本章后面描述。

8.2. 后备缓存Lookaside Caches

设备驱动程序通常会反复分配许多相同大小的对象。鉴于内核已经维护了一组所有对象都是相同大小的内存池，为什么不为这些高容量对象添加一些特殊的池呢？实际上，内核确实实现了一个创建这种池的设施，通常被称为侧视缓存。设备驱动程序通常不会表现出需要使用侧视缓存的内存行为，但可能会有例外；Linux 2.6 中的 USB 和 SCSI 驱动程序使用缓存。

Linux 内核中的缓存管理器有时被称为“slab 分配器”。因此，它的函数和类型在 <linux/slab.h> 中声明。slab 分配器实现了一种类型为 kmem_cache_t 的缓存；它们通过调用 kmem_cache_create 创建：

```c
kmem_cache_t *kmem_cache_create(const char *name, size_t
size,
                                size_t offset,
                                unsigned long flags,
                                void (*constructor)(void
*, kmem_cache_t *,
unsigned long flags),
                                void (*destructor)(void
*, kmem_cache_t *,
unsigned long flags));
```

该函数创建一个新的缓存对象，可以承载任意数量的内存区域，所有的内存区域都是相同的大小，由 size 参数指定。name 参数与此缓存关联，并作为可用于跟踪问题的管理信息；通常，它被设置为缓存的结构类型的名称。缓存保留指向名称的指针，而不是复制它，所以驱动程序应该传入一个指向静态存储中的名称的指针（通常名称只是一个字面字符串）。名称不能包含空格。

offset 是页面中第一个对象的偏移量；它可以用来确保分配的对象具有特定的对齐，但你最有可能使用 0 来请求默认值。flags 控制如何进行分配，是以下标志的位掩码：

- SLAB_NO_REAP 设置此标志可以保护缓存不被在系统寻找内存时减少。通常设置此标志是一个坏主意；避免不必要地限制内存分配器的行动自由是很重要的。
- SLAB_HWCACHE_ALIGN 此标志要求每个数据对象对齐到一个缓存行；实际对齐取决于主机平台的缓存布局。如果你的缓存包含在 SMP 机器上频繁访问的项目，这个选项可能是一个好选择。然而，为了实现缓存行对齐所需的填充可能会浪费大量的内存。
- SLAB_CACHE_DMA 此标志要求每个数据对象在 DMA 内存区域分配。

还有一组标志可以在调试缓存分配时使用；详见 mm/slab.c。然而，通常这些标志在用于开发的系统上通过内核配置选项全局设置。

函数的 constructor 和 destructor 参数是可选的函数（但没有 constructor 就不能有 destructor）；前者可以用来初始化新分配的对象，后者可以用来在对象的内存被释放回系统整体之前“清理”对象。

构造函数和析构函数可能很有用，但你应该记住一些约束。当为一组对象分配内存时，会调用构造函数；因为该内存可能包含多个对象，所以可能会多次调用构造函数。你不能假设构造函数会作为分配对象的直接效果被调用。同样，析构函数可以在未知的未来某个时间被调用，而不是在对象被释放后立即被调用。根据是否传递了 SLAB_CTOR_ATOMIC 标志（其中 CTOR 是构造函数的缩写），构造函数和析构函数可能被允许或不被允许睡眠。

为了方便，程序员可以使用同一个函数作为构造函数和析构函数；当被调用者是构造函数时，slab 分配器总是传递 SLAB_CTOR_CONSTRUCTOR 标志。

一旦创建了一个对象的缓存，你可以通过调用 kmem_cache_alloc 从中分配对象：

```c
void *kmem_cache_alloc(kmem_cache_t *cache, int flags);
```

这里，cache 参数是你之前创建的缓存；flags 与你传递给 kmalloc 的相同，如果 kmem_cache_alloc 需要自己去分配更多的内存，会参考这些标志。

要释放一个对象，使用 kmem_cache_free：

```c
void kmem_cache_free(kmem_cache_t *cache, const void
*obj);
```

当驱动程序代码完成缓存的使用，通常是在模块被卸载时，它应该释放其缓存，如下所示：

```c
int kmem_cache_destroy(kmem_cache_t *cache);
```

只有当从缓存分配的所有对象都已返回到它时，销毁操作才会成功。因此，模块应该检查 kmem_cache_destroy 的返回状态；失败表示模块内存在某种形式的内存泄漏（因为一些对象已经被丢弃）。

使用侧视缓存的一个附带好处是，内核维护了缓存使用的统计信息。这些统计信息可以从 /proc/slabinfo 获取。

8.2.1. 一个基于 Slab 缓存的 scull: scullc

下面是一个例子。scullc 是 scull 模块的精简版本，只实现了裸设备——持久内存区域。与使用 kmalloc 的 scull 不同，scullc 使用内存缓存。量子的大小可以在编译时和加载时修改，但不能在运行时修改——这将需要创建一个新的内存缓存，我们不想处理这些不必要的细节。

scullc 是一个完整的示例，可以用来尝试 slab 分配器。它与 scull 的区别只在于几行代码。首先，我们必须声明我们自己的 slab 缓存：

```c
/* 声明一个缓存指针：用它来处理所有设备 */

kmem_cache_t *scullc_cache;
```

在模块加载时，创建 slab 缓存的处理方式如下：

```c
/* scullc_init: 为我们的量子创建一个缓存 */

scullc_cache = kmem_cache_create("scullc",
scullc_quantum,

        0, SLAB_HWCACHE_ALIGN, NULL, NULL); /* 没有
ctor/dtor */

if (!scullc_cache) {

    scullc_cleanup( );

    return -ENOMEM;

}
```

这是它如何分配内存量子：

```c
/* 使用内存缓存分配一个量子 */

if (!dptr->data[s_pos]) {

    dptr->data[s_pos] = kmem_cache_alloc(scullc_cache,
GFP_KERNEL);

    if (!dptr->data[s_pos])

        goto nomem;

    memset(dptr->data[s_pos], 0, scullc_quantum);

}
```

这些行释放内存：

```c
for (i = 0; i < qset; i++)
if (dptr->data[i])
        kmem_cache_free(scullc_cache, dptr->data[i]);
```

最后，在模块卸载时，我们必须将缓存返回给系统：

```c
/* scullc_cleanup: 释放我们的量子的缓存 */
if (scullc_cache)
    kmem_cache_destroy(scullc_cache);
```

从 scull 到 scullc 的主要区别是速度略有提高和内存使用更好。由于量子是从恰好大小合适的内存片段池中分配的，所以它们在内存中的放置尽可能密集，与 scull 的量子相反，后者会带来不可预测的内存碎片化。

8.2.2. 内存池Memory Pools

在内核中有些地方，内存分配不能失败。为了在这些情况下保证分配，内核开发人员创建了一个称为内存池（或“mempool”）的抽象。内存池实际上只是一种侧视缓存的形式，它试图始终保持一份可用于紧急情况的空闲内存列表。

内存池有一个类型为 mempool_t（在 <linux/mempool.h> 中定义）；你可以用 mempool_create 创建一个：

```c
mempool_t *mempool_create(int min_nr,
                          mempool_alloc_t *alloc_fn,
                          mempool_free_t *free_fn,
                          void *pool_data);
```

min_nr 参数是池应始终保持的已分配对象的最小数量。实际的对象分配和释放由 alloc_fn 和 free_fn 处理，它们有这些原型：

```c
typedef void *(mempool_alloc_t)(int gfp_mask, void
*pool_data);

typedef void (mempool_free_t)(void *element, void
*pool_data);
```

mempool_create 的最后一个参数（pool_data）被传递给 alloc_fn 和 free_fn。如果需要，你可以编写专门的函数来处理 mempools 的内存分配。然而，通常，你只是想让内核 slab 分配器为你处理这个任务。有两个函数（mempool_alloc_slab 和 mempool_free_slab）执行内存池分配原型和 kmem_cache_alloc 和 kmem_cache_free 之间的阻抗匹配。因此，设置内存池的代码通常看起来像下面这样：

```c
cache = kmem_cache_create(. . .);

pool = mempool_create(MY_POOL_MINIMUM,

                      mempool_alloc_slab,
mempool_free_slab,

                      cache);
```

一旦创建了池，就可以用以下方式分配和释放对象：

```c
void *mempool_alloc(mempool_t *pool, int gfp_mask);

void mempool_free(void *element, mempool_t *pool);
```

当创建 mempool 时，将调用分配函数足够多的次数来创建一组预分配的对象。此后，mempool_alloc 的调用试图从分配函数获取额外的对象；如果该分配失败，将返回一个预分配的对象（如果还有的话）。当一个对象用 mempool_free 释放时，如果预分配的对象数量当前低于最小值，它将被保留在池中；否则，它将被返回给系统。

你可以使用以下方法调整 mempool 的大小：

```c
int mempool_resize(mempool_t *pool, int new_min_nr, int
gfp_mask);
```

如果成功，此调用将调整池的大小，使其至少有 new_min_nr 个对象。如果你不再需要内存池，用以下方法将其返回给系统：

```c
void mempool_destroy(mempool_t *pool);
```

在销毁 mempool 之前，你必须返回所有已分配的对象，否则会导致内核 oops。

如果你正在考虑在你的驱动程序中使用 mempool，请记住一件事：mempools 分配一块内存，这块内存位于一个列表中，处于空闲状态，无法用于任何实际用途。使用 mempools 很容易消耗大量的内存。在几乎所有情况下，首选的替代方案是不使用 mempool，而是简单地处理分配失败的可能性。如果你的驱动程序有任何方式可以响应分配失败，而不会危及系统的完整性，那就用那种方式。在驱动代码中使用 mempools 应该是罕见的。

8.3. get_free_page 和其friends

如果一个模块需要分配大块的内存，通常最好使用面向页面的技术。请求整个页面也有其他优点，这些优点将在第15章中介绍。

为了分配页面，可以使用以下函数：

```c
get_zeroed_page(unsigned int flags);
```

返回一个新页面的指针，并用零填充页面。

```c
__get_free_page(unsigned int flags);
```

与 get_zeroed_page 类似，但不清除页面。

```c
__get_free_pages(unsigned int flags, unsigned int order);
```

分配并返回一个指向内存区域第一个字节的指针，该区域可能是几个（物理连续的）页面长，但不将该区域归零。

flags 参数的工作方式与 kmalloc 相同；通常使用 GFP_KERNEL 或 GFP_ATOMIC，可能还会添加 `__GFP_DMA` 标志（用于可以用于 ISA 直接内存访问操作的内存）或 `__GFP_HIGHMEM`（当可以使用高内存时）。`* order` 是你请求或释放的页面数量的

以二为底的对数（即，log2N）。例如，如果你想要一个页面，order 是 0；如果你请求八个页面，order 是 3。如果 order 太大（没有那么大的连续区域可用），页面分配失败。get_order 函数，它接受一个整数参数，可以用来从大小（必须是二的幂）中提取出主机平台的 order。order 的最大允许值是 10 或 11（对应 1024 或 2048 页），取决于架构。然而，除非是刚刚启动的系统并且有大量的内存，否则 order-10 分配成功的机会很小。

如果你好奇，/proc/buddyinfo 会告诉你系统上每个内存区域有多少个每个 order 的块可用。

当程序完成页面使用后，可以使用以下函数之一释放它们。第一个函数是一个宏，它回退到第二个函数：

```c
void free_page(unsigned long addr);

void free_pages(unsigned long addr, unsigned long order);
```

如果你试图释放与你分配的页面数量不同的页面，内存映射会变得混乱，系统在以后的时间会遇到问题。

值得强调的是，`__get_free_pages` 和其他函数可以在任何时候被调用，遵循我们为 kmalloc 看到的相同规则。在某些情况下，这些函数可能无法分配内存，特别是当使用 GFP_ATOMIC 时。因此，调用这些分配函数的程序必须准备好处理分配失败。

虽然当没有可用内存时 kmalloc(GFP_KERNEL) 有时会失败，但内核会尽其所能满足分配请求。因此，通过分配过多的内存很容易降低系统的响应性。例如，你可以通过向 scull 设备中推送过多的数据来使计算机崩溃；系统在尝试尽可能多地交换出来以满足 kmalloc 请求时开始爬行。由于每个资源都被不断增长的设备吸收，计算机很快就变得无法使用；此时，你甚至无法启动新的进程来处理问题。我们在 scull 中没有解决这个问题，因为它只是一个示例模块，而不是一个可以放入多用户系统的真实工具。然而，作为一个程序员，你必须非常小心，因为一个模块是特权代码，可以在系统中打开新的安全漏洞（最可能的是像刚才概述的那样的拒绝服务漏洞）。

- 虽然 alloc_pages（稍后描述）应该真正用于分配高内存页面，但由于我们在第15章之前无法真正深入讨论，所以有些原因我们无法真正深入讨论。

8.3.1. 一个使用整页的 scull: scullp

为了真正测试页面分配，我们已经发布了 scullp 模块以及其他示例代码。它是一个简化的 scull，就像前面介绍的 scullc 一样。 scullp 分配的内存量是整个页面或页面集：scullp_order 变量默认为 0，但可以在编译或加载时更改。

以下行显示了它如何分配内存：

```c
/* 这是单个量子的分配 */

if (!dptr->data[s_pos]) {

    dptr->data[s_pos] =

        (void *)__get_free_pages(GFP_KERNEL, dptr-
>order);

    if (!dptr->data[s_pos])

        goto nomem;

    memset(dptr->data[s_pos], 0, PAGE_SIZE << dptr-
>order);

}
```

在 scullp 中释放内存的代码如下：

```c
/* 这段代码释放了整个量子集 */
for (i = 0; i < qset; i++)
    if (dptr->data[i])
        free_pages((unsigned long)(dptr->data[i]),
                   dptr->order);
```

在用户级别，感知到的主要区别是速度的提高和更好的内存使用，因为没有内存的内部碎片。我们进行了一些测试，从 scull0 复制 4 MB 到 scull1，然后从 scullp0 复制到 scullp1；结果显示内核空间处理器使用率有所提高。

性能的提高并不显著，因为 kmalloc 设计得很快。页面级分配的主要优点实际上并不是速度，而是更有效的内存使用。按页面分配不会浪费内存，而使用 kmalloc 由于分配粒度会浪费不可预测的内存量。

但是，`__get_free_page` 函数的最大优点是获得的页面完全属于你，理论上，你可以通过适当地调整页面表将页面组装成一个线性区域。例如，你可以允许用户进程 mmap 获取的单个无关页面的内存区域。我们在第15章中讨论了这种操作，我们展示了如何提供内存映射，这是 scull 无法提供的。

8.3.2. alloc_pages 接口

为了完整性，我们介绍另一个内存分配的接口，尽管我们在第15章之后才准备使用它。现在，只需说 struct page 是一个内部内核结构，描述了一个内存页面。我们将看到，在内核中有许多地方需要使用页面结构；它们在任何可能处理高内存的情况下都特别有用，高内存在内核空间中没有常量地址。

Linux 页面分配器的真正核心是一个名为 alloc_pages_node 的函数：

```c
struct page *alloc_pages_node(int nid, unsigned int
flags,
                              unsigned int order);
```

这个函数也有两个变体（它们只是宏）；这些是你最有可能使用的版本：

```c
struct page *alloc_pages(unsigned int flags, unsigned int
order);

struct page *alloc_page(unsigned int flags);
```

核心函数 alloc*pages_node 接受三个参数。nid 是应该分配内存的 NUMA 节点 ID*，*flags 是通常的 GFP* 分配标志，order 是分配的大小。返回值是指向描述分配内存的（可能有很多）页面结构的第一个指针，或者，像往常一样，失败时为 NULL。

alloc_pages 简化了情况，通过在当前的 NUMA 节点上分配内存（它调用 alloc_pages_node，将 numa_node_id 的返回值作为 nid 参数）。当然，alloc_page 省略了 order 参数，并分配了一个单独的页面。

要释放以这种方式分配的页面，你应该使用以下之一：

```c
void __free_page(struct page *page);

void __free_pages(struct page *page, unsigned int order);

void free_hot_page(struct page *page);

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

如果你具有关于单个页面的内容是否可能驻留在处理器缓存中的特定知识，你应该使用 free_hot_page（对于缓存驻留页面）或 free_cold_page 将该信息传达给内核。这些

信息帮助内存分配器优化其在系统中的内存使用。

8.3.3. vmalloc 和 其friends

我们要展示的下一个内存分配函数是 vmalloc，它在虚拟地址空间中分配一个连续的内存区域。虽然这些页面在物理内存中并不连续（每个页面都是通过单独调用 alloc_page 获取的），但内核将它们视为连续的地址范围。如果发生错误，vmalloc 返回 0（NULL 地址），否则，它返回一个指向至少为 size 大小的线性内存区域的指针。

- NUMA（非均匀内存访问）计算机是多处理器系统，其中内存是“本地”的，特定的处理器组（“节点”）可以访问。访问本地内存比访问非本地内存快。在这样的系统上，正确地在节点上分配内存是很重要的。然而，驱动程序作者通常不需要担心 NUMA 问题。
- 以下是 vmalloc 的函数签名：

```c
void *vmalloc(unsigned long size);
```

- 你可以使用以下函数释放通过 vmalloc 分配的内存：

```
void vfree(void *addr);
```

我们在这里描述 vmalloc，因为它是 Linux 内存分配机制的基础之一。然而，我们应该注意，在大多数情况下，不鼓励使用 vmalloc。从 vmalloc 获取的内存稍微低效一些，而且，在某些架构上，为 vmalloc 留出的地址空间相对较小。使用 vmalloc 的代码如果提交到内核中可能会受到冷淡的接待。如果可能，你应该直接处理单个页面，而不是试图用 vmalloc 来平滑事情。

话虽如此，让我们看看 vmalloc 是如何工作的。函数及其相关函数（ioremap，它并不严格是一个分配函数，将在本节后面讨论）的原型如下：

```c
#include <linux/vmalloc.h>

void *vmalloc(unsigned long size);

void vfree(void * addr);

void *ioremap(unsigned long offset, unsigned long size);

void iounmap(void * addr);
```

值得强调的是，由 kmalloc 和 `_get_free_pages` 返回的内存地址也是虚拟地址。它们的实际值在被用来寻址物理内存之前仍然被 MMU（内存管理单元，通常是 CPU 的一部分）处理。vmalloc 在使用硬件方面并无不同，而在于内核如何执行分配任务。

由 kmalloc 和 `__get_free_pages` 使用的（虚拟）地址范围具有一对一的物理内存映射，可能被常量 `PAGE_OFFSET` 值偏移；这些函数不需要修改该地址范围的页面表。另一方面，由 vmalloc 和 ioremap 使用的地址范围完全是合成的，每个分配都通过适当地设置页面表来构建（虚拟）内存区域。

通过比较分配函数返回的指针可以感知到这种差异。在某些平台（例如，x86）上，vmalloc 返回的地址刚好超过 kmalloc 使用的地址。在其他平台（例如，MIPS，IA-64 和 x86_64）上，它们属于完全不同的地址范围。可用于 vmalloc 的地址在 VMALLOC_START 到 VMALLOC_END 的范围内。这两个符号都在 <asm/pgtable.h> 中定义。

由 vmalloc 分配的地址不能在微处理器之外使用，因为它们只有在处理器的 MMU 之上才有意义。当驱动程序需要一个真正的物理地址（如 DMA 地址，由外围硬件用来驱动系统的总线）时，你不能轻易地使用 vmalloc。当你为只存在于软件中的大型顺序缓冲区分配内存时，调用 vmalloc 是合适的。

需要注意的是，vmalloc 的开销比 `__get_free_pages` 大，因为它既要检索内存，又要构建页面表。因此，调用 vmalloc 来分配只有一页的内存是没有意义的。

内核中使用 vmalloc 的一个函数的例子是 create_module 系统调用，它使用 vmalloc 来获取正在创建的模块的空间。模块的代码和数据稍后使用 copy_from_user 复制到分配的空间。这样，模块看起来就加载到了连续的内存中。你可以通过查看

/proc/kallsyms 来验证，由模块导出的内核符号位于与内核本身导出的符号不同的内存范围中。

使用 vmalloc 分配的内存通过 vfree 释放，就像 kfree 释放由 kmalloc 分配的内存一样。

像 vmalloc 一样，ioremap 构建新的页面表；然而，与 vmalloc 不同的是，它实际上并没有分配任何内存。ioremap 的返回值是一个特殊的虚拟地址，可以用来访问指定的物理地址范围；最终通过调用 iounmap 释放获得的虚拟地址。

ioremap 最有用的是将 PCI 缓冲区的（物理）地址映射到（虚拟）内核空间。例如，它可以用来访问 PCI 视频设备的帧缓冲区；这些缓冲区通常映射在高物理地址上，超出了内核在启动时为其构建页面表的地址范围。PCI 问题在第12章中详细解释。

值得注意的是，为了可移植性，你不应该直接访问 ioremap 返回的地址，就好像它们是指向内存的指针一样。相反，你应该始终使用 readb 和第9章中介绍的其他 I/O 函数。这个要求适用于一些平台，如 Alpha，由于 PCI 规范和 Alpha 处理器在数据传输方式上的差异，无法直接将 PCI 内存区域映射到处理器地址空间。

ioremap 和 vmalloc 都是面向页面的（它们通过修改页面表来工作）；因此，重新定位或分配的大小被四舍五入到最近的页面边界。ioremap 通过“向下取整”要重新映射的地址，并返回到第一个重新映射的页面的偏移量，来模拟一个未对齐的映射。

vmalloc 的一个小缺点是它不能在原子上下文中使用，因为它内部使用 kmalloc(GFP_KERNEL) 来获取页面表的存储，因此可能会睡眠。这不应该是一个问题——如果 `__get_free_page` 的使用对于中断处理程序来说还不够好，那么软件设计需要一些清理。

- 一些架构定义了一系列的“虚拟”地址，这些地址被保留用于寻址物理内存。当这种情况发生时，Linux 内核会利用这个特性，内核和 `__get_free_pages` 的地址都位于这些内存范围之一。这种差异对于设备驱动程序和其他不直接涉及内存管理内核子系统的代码是透明的。这意味着，除非你正在编写涉及到内存管理的底层代码，否则你不需要关心这些细节。你只需要知道，当你从 kmalloc 或 `__get_free_pages` 获取内存时，你得到的是可以直接使用的虚拟地址。

8.3.4. 一个使用虚拟地址的 scull：scullv

在 scullv 模块中提供了使用 vmalloc 的示例代码。像 scullp 一样，这个模块是 scull 的精简版本，它使用不同的分配函数来获取设备存储数据的空间。

该模块一次分配 16 页的内存。分配是以大块进行的，以实现比 scullp 更好的性能，并展示其他分配技术无法实现的长时间操作。使用 `__get_free_pages` 分配多于一页的页面容易失败，即使成功，也可能很慢。如我们之前看到的，vmalloc 在分配多个页面时比其他函数快，但在检索单个页面时由于页面表构建的开销而稍慢。scullv 的设计类似于 scullp。order 指定每次分配的“顺序”，默认为 4。scullv 和 scullp 的唯一区别在于分配管理。这些行使用 vmalloc 获取新的内存：

```c
/* 使用虚拟地址分配一个量子 */
if (!dptr->data[s_pos]) {
    dptr->data[s_pos] =
        (void *)vmalloc(PAGE_SIZE << dptr->order);
    if (!dptr->data[s_pos])
        goto nomem;
    memset(dptr->data[s_pos], 0, PAGE_SIZE << dptr-
>order);
}
```

这些行释放内存：

```c
/* 释放量子集 */
for (i = 0; i < qset; i++)
    if (dptr->data[i])
        vfree(dptr->data[i]);
```

如果你在启用调试的情况下编译这两个模块，你可以通过读取它们在 /proc 中创建的文件来查看它们的数据分配。这个快照是在一个 x86_64 系统上拍摄的：

```
salma% cat /tmp/bigfile > /dev/scullp0; head -5 /proc/scul
lpmem

Device 0: qset 500, order 0, sz 1535135

  item at 000001001847da58, qset at 000001001db4c000

       0:1001db56000

       1:1003d1c7000

salma% cat /tmp/bigfile > /dev/scullv0; head -5 /proc/scul
lvmem

Device 0: qset 500, order 4, sz 1535135

  item at 000001001847da58, qset at 0000010013dea000

       0:ffffff0001177000

       1:ffffff0001188000
```

以下输出，来自一个 x86 系统：

```
rudo% cat /tmp/bigfile > /dev/scullp0; head -5 /proc/scull
pmem

Device 0: qset 500, order 0, sz 1535135

  item at ccf80e00, qset at cf7b9800

       0:ccc58000

       1:cccdd000

rudo%  cat /tmp/bigfile > /dev/scullv0; head -5 /proc/scul
lvmem

Device 0: qset 500, order 4, sz 1535135

  item at cfab4800, qset at cf8e4000

       0:d087a000

       1:d08d2000
```

这些值显示了两种不同的行为。在 x86_64 上，物理地址和虚拟地址映射到完全不同的地址范围（0x100 和 0xffffff00），而在 x86 计算机上，vmalloc 返回的虚拟地址刚好在用于物理内存映射的地址之上。

8.4. 每个CPU 的变量

每个 CPU 变量是 2.6 内核的一个有趣特性。当你创建一个每个 CPU 的变量时，系统上的每个处理器都会得到该变量的一个副本。这可能看起来很奇怪，但它有其优点。访问每个 CPU 的变量几乎不需要锁定，因为每个处理器都使用自己的副本。每个 CPU 的变量也可以保留在各自处理器的缓存中，这对于频繁更新的数量可以显著提高性能。

一个很好的每个 CPU 变量使用的例子可以在网络子系统中找到。内核维护了无数的计数器，跟踪接收了多少种类型的数据包；这些计数器每秒可以更新数千次。而不是处理缓存和锁定问题，网络开发人员将统计计数器放入每个 CPU 的变量中。现在的更新是无锁的，速度快。在用户空间很少请求查看计数器的值的情况下，只需简单地加上每个处理器的版本并返回总数。

每个 CPU 变量的声明可以在 <linux/percpu.h> 中找到。要在编译时创建一个每个 CPU 的变量，使用这个宏：

```c
DEFINE_PER_CPU(type, name);
```

如果变量（被称为 name）是一个数组，将维度信息与类型一起包含。因此，一个每个 CPU 的三个整数的数组将用以下方式创建：

```c
DEFINE_PER_CPU(int[3], my_percpu_array);
```

每个 CPU 的变量可以在没有显式锁定的情况下操作——几乎可以。记住，2.6 内核是可抢占的；如果一个处理器在修改每个 CPU 变量的关键部分被抢占，那就不好了。如果你的进程在访问每个 CPU 变量的过程中被移动到另一个处理器，那也不好。因此，你必须显式使用 get_cpu_var 宏来访问当前处理器的给定变量的副本，并在完成时调用 put_cpu_var。调用 get_cpu_var 返回当前处理器版本的变量的 lvalue，并禁用抢占。由于返回了一个 lvalue，所以可以直接赋值或操作。例如，网络代码中的一个计数器用这两个语句增加：

```c
get_cpu_var(sockets_in_use)++;

put_cpu_var(sockets_in_use);
```

你可以使用以下方式访问另一个处理器的变量副本：

```c
per_cpu(variable, int cpu_id);
```

如果你编写的代码涉及到处理器访问彼此的每个 CPU 的变量，你当然需要实现一个锁定方案，使得这种访问是安全的。

也可以动态分配每个 CPU 的变量。这些变量可以用以下方式分配：

```c
void *alloc_percpu(type);

void *__alloc_percpu(size_t size, size_t align);
```

在大多数情况下，alloc_percpu 可以完成工作；在需要特定对齐的情况下，你可以调用 `__alloc_percpu`。在任何情况下，每个 CPU 的变量都可以用 free_percpu 返回给系统。访问动态分配的每个 CPU 的变量是通过 per_cpu_ptr 完成的：

```c
per_cpu_ptr(void *per_cpu_var, int cpu_id);
```

这个宏返回一个指向与给定 cpu_id 对应的 per_cpu_var 版本的指针。如果你只是读取另一个 CPU 的变量版本，你可以解引用那个指针并完成它。然而，如果你正在操作当前处理器的版本，你可能需要确保你不能被移出那个处理器。如果你访问每个 CPU 变量的全部都在持有自旋锁的情况下发生，那就没问题了。然而，通常情况下，你需要使用 get_cpu 来阻止抢占，同时处理变量。因此，使用动态每个 CPU 变量的代码往往看起来像这样：

```c
int cpu;

cpu = get_cpu();

ptr = per_cpu_ptr(per_cpu_var, cpu);

/* work with ptr */

put_cpu();
```

当使用编译时每个 CPU 的变量时，get_cpu_var 和 put_cpu_var 宏会处理这些细节。动态每个 CPU 的变量需要更明确的保护。

每个 CPU 的变量可以被导出到模块，但你必须使用宏的特殊版本：

```c
EXPORT_PER_CPU_SYMBOL(per_cpu_var);

EXPORT_PER_CPU_SYMBOL_GPL(per_cpu_var);
```

要在模块中访问这样的变量，声明它：

```c
DECLARE_PER_CPU(type, name);
```

使用 DECLARE_PER_CPU（而不是 DEFINE_PER_CPU）告诉编译器正在进行外部引用。

如果你想使用每个 CPU 的变量来创建一个简单的整数计数器，可以看看 <linux/percpu_counter.h> 中的预制实现。最后，注意一些架构的每个 CPU 变量的地址空间有限。如果你在代码中创建每个 CPU 的变量，你应该尽量保持它们小。

8.5. 获得大量缓冲

正如我们在前面的部分中提到的，大型连续内存缓冲区的分配容易失败。系统内存会随着时间的推移而碎片化，很可能真正大的内存区域根本就不可用。由于通常有方法可以在没有大型缓冲区的情况下完成工作，内核开发人员并没有将使大型分配工作作为高优先级任务。在你试图获取大型内存区域之前，你真的应该考虑其他的选择。迄今为止，执行大型 I/O 操作的最好方法是通过散列/聚集操作，我们将在第 15 章的“散列-聚集映射”部分进行讨论。

8.5.1. 在启动时获得专用的缓冲

如果你真的需要一个物理连续的巨大缓冲区，最好的方法通常是通过在启动时请求内存来分配它。启动时分配是唯一一种可以在绕过 `__get_free_pages` 对缓冲区大小施加的限制的同时检索连续内存页的方法，无论是在允许的最大大小还是大小选择的限制方面。在启动时分配内存是一种“脏”技术，因为它通过预留私有内存池来绕过所有内存管理策略。这种技术不优雅且不灵活，但它也最不容易失败。不用说，模块不能在启动时分配内存；只有直接链接到内核的驱动程序才能做到这一点。

启动时分配的一个显著问题是，它对于普通用户来说并不是一个可行的选项，因为这种机制只适用于链接在内核映像中的代码。使用这种类型分配的设备驱动程序只能通过重建内核并重新启动计算机来安装或替换。

当内核启动时，它可以访问系统中所有可用的物理内存。然后，它通过调用该子系统的初始化函数来初始化每个子系统，允许初始化代码通过减少留给正常系统操作的 RAM 量来分配一个用于私有使用的内存缓冲区。

启动时的内存分配是通过调用以下函数之一来执行的：

```c
#include <linux/bootmem.h>

void *alloc_bootmem(unsigned long size);

void *alloc_bootmem_low(unsigned long size);

void *alloc_bootmem_pages(unsigned long size);

void *alloc_bootmem_low_pages(unsigned long size);
```

这些函数分配整个页面（如果它们以 `_pages` 结尾）或非页面对齐的内存区域。分配的内存可能是高内存，除非使用了 `_low` 版本之一。如果你正在为设备驱动程序分配这个缓冲区，你可能想要用它进行 DMA 操作，而这并不总是可以用高内存来完成的；因此，你可能想要使用 `_low` 变体之一。

很少释放在启动时分配的内存；如果你想要的话，你几乎肯定无法再得到它。然而，有一个接口可以释放这个内存：

```c
void free_bootmem(unsigned long addr, unsigned long
size);
```

请注意，以这种方式释放的部分页面不会返回给系统——但是，如果你正在使用这种技术，你可能已经分配了相当多的整个页面。

如果你必须使用启动时分配，你需要将你的驱动程序直接链接到内核。有关如何做到这一点的更多信息，请参阅内核源代码中 Documentation/kbuild 下的文件。

8.6. 快速参考

相关于内存分配的函数和符号是:

```
#include <linux/slab.h>
void *kmalloc(size_t size, int flags);
void kfree(void *obj);
```

内存分配的最常用接口.

```
#include <linux/mm.h>
GFP_USER
GFP_KERNEL
GFP_NOFS
GFP_NOIO
GFP_ATOMIC
```

控制内存分配如何进行的标志, 从最少限制的到最多的. GFP_USER 和 GFP_KERNEL 优先级允许当前进程被置为睡眠来满足请求. GFP_NOFS 和 GFP_NOIO 禁止文件系统操作和所有的 I/O 操作, 分别地, 而 GFP_ATOMIC 分配根本不能睡眠.

```
__GFP_DMA
__GFP_HIGHMEM
__GFP_COLD
__GFP_NOWARN
__GFP_HIGH
__GFP_REPEAT
__GFP_NOFAIL
__GFP_NORETRY
```

这些标志修改内核的行为, 当分配内存时.

```
#include <linux/malloc.h>
kmem_cache_t *kmem_cache_create(char *name, size_t size,
size_t offset, unsigned long flags, constructor(),
destructor( ));
int kmem_cache_destroy(kmem_cache_t *cache);
```

创建和销毁一个 slab 缓存. 这个缓存可被用来分配几个相同大小的对象.

```
SLAB_NO_REAP
SLAB_HWCACHE_ALIGN
SLAB_CACHE_DMA
```

在创建一个缓存时可指定的标志.

```
SLAB_CTOR_ATOMIC
SLAB_CTOR_CONSTRUCTOR
```

分配器可用传递给构造函数和析构函数的标志.

```
void *kmem_cache_alloc(kmem_cache_t *cache, int flags);
void kmem_cache_free(kmem_cache_t *cache, const void
*obj);
```

从缓存中分配和释放一个单个对象. /proc/slabinfo 一个包含对 slab 缓存使用情况统计的虚拟文件.

```
#include <linux/mempool.h>
mempool_t *mempool_create(int min_nr, mempool_alloc_t
*alloc_fn, mempool_free_t *free_fn, void *data);
void mempool_destroy(mempool_t *pool);
```

创建内存池的函数, 它试图避免内存分配设备, 通过保持一个已分配项的"紧急列表".

```
void *mempool_alloc(mempool_t *pool, int gfp_mask);
void mempool_free(void *element, mempool_t *pool);
```

从(并且返回它们给)内存池分配项的函数.

```
unsigned long get_zeroed_page(int flags);
unsigned long __get_free_page(int flags);
unsigned long __get_free_pages(int flags, unsigned long
order);
```

面向页的分配函数. get_zeroed_page 返回一个单个的, 零填充的页. 这个调用的所有的其他版本不初始化返回页的内容.

```
int get_order(unsigned long size);
```

返回关联在当前平台的大小的分配级别, 根据 PAGE_SIZE. 这个参数必须是 2 的幂, 并且返回值至少是 0.

```
void free_page(unsigned long addr);
void free_pages(unsigned long addr, unsigned long order);
```

释放面向页分配的函数.

```
struct page *alloc_pages_node(int nid, unsigned int flags,
unsigned int order);
struct page *alloc_pages(unsigned int flags, unsigned int
order);
struct page *alloc_page(unsigned int flags);
```

Linux 内核中最底层页分配器的所有变体.

```
void __free_page(struct page *page);
void __free_pages(struct page *page, unsigned int order);
void free_hot_page(struct page *page);
```

使用一个 alloc_page 形式分配的页的各种释放方法.

```
#include <linux/vmalloc.h>
void * vmalloc(unsigned long size);
void vfree(void * addr);
#include <asm/io.h>
void * ioremap(unsigned long offset, unsigned long size);
void iounmap(void *addr);
```

分配或释放一个连续虚拟地址空间的函数. iormap 存取物理内存通过虚拟地址, 而 vmalloc 分配空闲页. 使用 ioreamp 映射的区是 iounmap 释放, 而从 vmalloc 获得的页使用 vfree 来释放.

```
#include <linux/percpu.h>
DEFINE_PER_CPU(type, name);
DECLARE_PER_CPU(type, name);
```

定义和声明每-CPU变量的宏.

```
per_cpu(variable, int cpu_id)
get_cpu_var(variable)
put_cpu_var(variable)
```

提供对静态声明的每-CPU变量存取的宏.

```
void *alloc_percpu(type);
void *__alloc_percpu(size_t size, size_t align);
void free_percpu(void *variable);
```

进行运行时分配和释放每-CPU变量的函数.

```
int get_cpu( );
void put_cpu( );
per_cpu_ptr(void *variable, int cpu_id)
```

get_cpu 获得对当前处理器的引用(因此, 阻止抢占和移动到另一个处理器)并且返回处理器的ID; put_cpu 返回这个引用. 为存取一个动态分配的每-CPU变量, 用应当被存取版本所在的 CPU 的 ID 来使用 per_cpu_ptr. 对一个动态的每-CPU 变量当前 CPU 版本的操作, 应当用对 get_cpu 和 put_cpu 的调用来包围.

```
#include <linux/bootmem.h>
void *alloc_bootmem(unsigned long size);
void *alloc_bootmem_low(unsigned long size);
void *alloc_bootmem_pages(unsigned long size);
void *alloc_bootmem_low_pages(unsigned long size);
void free_bootmem(unsigned long addr, unsigned long size);
```

在系统启动时进行分配和释放内存的函数(只能被直接连接到内核中去的驱动使用)