第 7 章 获取内存

到目前为止，我们总是用 *kmalloc* 和 *kfree* 来进行内存分配。当然，只用这些函数的确是管理内存的捷径。本章将会介绍其他一些内存分配技术。但我们目前并不关心不同的体系结构实际上是如何进行内存管理的。因为内核为设备驱动程序提供了一致的接口，本章的模块都不必涉及分段，分页等问题。另外，本章我也不会介绍内存管理的内部细节，这些问题将留到第 13 章“*Mmap 和 DMA*”的“Linux 的内存管理”一节讨论。

kmalloc 函数的内幕

kmalloc 内存分配引擎功能强大，由于和 *malloc* 函数很相似，很容易就可以学会。这个函数运行得很快－－除非它被阻塞－－它不清零它获得的内存空间；分配给它的区域仍存放着原有的数据。在下面几节，我会详细介绍 *kmalloc* 函数，你可以将它和我后面要介绍的一些内存分配技术作个比较。

优先权参数

kmalloc 函数的第一个参数是 size(大小)，我留在下个小节介绍。第二个参数，是优先权，更有意思，因为它会使得 *kmalloc* 函数在寻找空闲页较困难时改变它的行为。

最常用的优先权是 **GFP_KERNEL**，它的意思是该内存分配(内部是通过调用 *get_free_pages* 来实现的，所以名字中带 **GFP**)是由运行在内核态的进程调用的。也就是说，调用它的函数属于某个进程的，使用 **GFP_KERNEL** 优先权允许 *kmalloc* 函数在系统空闲内存低于水平线 **min_free_pages** 时延迟分配函数的返回。当空闲内存太少时，*kmalloc* 函数会使当前进程进入睡眠，等待空闲页的出现。

新的页面可以通过以下几种途径获得。一种方法是换出其他页；因为对换需要时间，进程会等待它完成，这时内核可以调度执行其他的任务。因此，每个调用 **kmalloc(GFP_KERNEL)**的内核函数都应该是可重入的。关于可重入的更多细节可见第 5 章“*字符设备驱动程序的扩展操作*”的“编写可重入的代码”一节。

并非使用 **GFP_KERNEL** 优先权后一定正确；有时 *kmalloc* 是在进程上下文之外调用的－－比如，在中断处理，任务队列处理和内核定时器处理时发生。这些情况下，**current** 进程就不应该进入睡眠，这时应该就使用优先权 **GFP_ATOMIC**。原子性(atomic)的内存分配允许使用内存的空闲位，而与 **min_free_pages** 值无关。实际上，这个最低水平线值的存在就是为了能满足原子性的请求。但由于内核并不允许通过换出数据或缩减文件系统缓冲区来满足这种分配请求，所以必须还有一些真正可以获得的空闲内存。

为 *kmalloc* 还定义了其他一些优先权，但都不经常使用，其中一些只在内部的内存管理算法中使用。另一个值的注意的优先权是 **GFP_NFS**，它会使得 NFS 文件系统缩减空闲列表

到 **min_free_pages** 值以下。显然，为使驱动程序“更快”而用 **GFP_NFS** 优先权取代 **GFP_KERNEL** 优先权会降低整个系统的性能。

除了这些常用的优先权，*kmalloc* 还可以识别一个位域：**GFP_DMA**。**GFP_DMA** 标志位要和 **GFP_KERNEL** 和 **GFP_ATOMIC** 优先权一起使用来分配用于直接内存访问(DMA)的内存页。我们将在第 13 章的“直接内存访问”一节讨论如何使用这个标志位。

size 参数

系统物理内存的管理是由内核负责的，*物理*内存只能按页大小进行分配。这就需要一个面向页的分配技术以取得计算机内存管理上最大的灵活性。类似 *malloc* 函数的简单的线性的分配技术不再有效了；在象 Unix 内核这样的面向页的系统中内存如果是线性分配的就很难维护。空洞的处理很快就会成为一个问题，会导致内存浪费，降低系统的性能。

Linux 是通过维护页面池来处理 *kmalloc* 的分配要求的，这样页面就可以很容易地放进或者取出页面池。为了能够满足超过 **PAGE_SIZE** 字节数大小的内存分配请求，*fs/kmalloc.c* 文件维护页面簇的列表。每个页面簇都存放着连续若干页，可用于 DMA 分配。在这里我不介绍底层的实现细节，因为内部的数据结构可以在不影响分配语义和驱动程序代码的前提下加以改变。事实上，2.1.38 版已经将 kmalloc 重新实现了。2.0 版的内存分配实现代码可以参见文件 *mm/malloc.c*，而新版的实现在文件 *mm/slab.c* 中。想了解 2.0 版实现的详情可参见第 16 章“*内核代码的物理布局*”的“分配和释放”一节。

Linux 所使用的分配策略的最终方案是，内核只能分配一些预定义的固定大小的字节数组。如果你申请任意大小的内存空间，那么很可能系统会多给你一点。

这些预定义的内存大小一般“稍小于 2 的某次方”(而在更新的实现中系统管理的内存大小恰好为 2 的各次方)。如果你能记住这一点，就可以更有效地使用内存了。例如，如果在 Linux 2.0 上你需要一个 2000 字节左右的缓冲区，你最好还是申请 2000 字节，而不要申请 2048 字节。在低与 2.1.38 版的内核中，申请恰好是 2 的幂次的内存空间是最糟糕的情况了－内核会分配两倍于你申请空间大小的内存给你。这也就是为什么在示例程序 *scull* 中每个单元(quantum)要用 4000 字节而不是 4096 字节的原因了。

你可以从文件 *mm/malloc.c*(或者 *mm/slab.c*)得到预定义的分配块大小的确切数值，但注意这些值可能在以后的版本中被改变。在当前的 2.0 版和 2.1 版的内核中，都可以用个小技巧－尽量分配小于 4K 字节的内存空间，但不能保证这种方法将来也是最优的。

无论如何，Linux2.0 中 *kmalloc* 函数可以分配的内存空间最大不能超过 32 个页－Alpha 上的 256KB 或者 Intel 和其他体系结构上的 128KB。2.1.38 版和更新的内核中这个上限是 128KB。如果你需要更多一些空间，那么有下面一些的更好的解决方法。

get_free_page 和相关函数

如果模块需要分配大块的内存，那使用面向页的分配技术会更好。请求整页还有其他一

些好处，后面第 13 章的“mmap 设备驱动程序操作”一节将会介绍。

分配页面可使用下面一些函数：

- *get_free_page* 返回指向新页面的指针并将页面清零。
- *__get_free_pages* 和 *get_free_page* 类似，但不清零页面。
- *__get_free_pages* 返回一个指向大小为几个页的内存区域的第一个字节位置的指针，但也不清零这段内存区域。
- *__get_dma_pages* 返回一个指向大小为几个页的内存区域的第一个字节位置的指针；这些页面在物理上是连续的，可用于 DMA 传输。

这些函数的原型在 Linux2.0 中定义如下：

```
unsigned long get_free_page(int priority);
unsigned long __get_free_page(int priority);
unsigned long __get_dma_pages(int priority, unsigned long order);
unsigned long __get_free_pages(int priority, unsigned long order, int dma);
```

实际上，除了*__get_free_pages*，这些函数或者是宏或者是最终调用了*__get_free_pages*的内联函数。

当程序使用完分配给它的页面，就应该调用下面的函数。下面的第一个函数是个宏，其中调用了第二个函数：

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

如果你希望代码在 1.2 版和 2.0 版的 Linux 上都能运行，那最好还是不要直接使用函数*__get_free_pages*，因为它的调用方式在这两个版本间修改过 2 次。只使用函数*get_free_page*(和*__get_free_page*)更安全更可移植，而且也足够了。

至于 DMA，由于 PC 平台设计上的一些“特殊性”，要正确寻址 ISA 卡还有些问题。我在第 13 章的“直接内存访问”一节中介绍 DMA 时，将只限于 2.0 版内核上的实现，以避免引入移植方面的问题。

分配函数中的**priority**参数和*kmalloc*函数中含义是一样的。*__get_free_pages*函数中的**dma**参数是零或非零；如果不是零，那么对分配的页面簇可以进行DMA传输。**order**是你请求分配或释放的内存空间相对 2 的幂次(即$\log_2 N$)。例如，如果需要 1 页，**order**为 0；需要 8 页，**order**为 3。如果**order**太大，分配就会失败。如果你释放的内存空间大小和分配得到的大小不同，那么有可能破坏内存映射。在Linux目前的版本中，**order**最大为 5(相当于 32 个页)。总之，**order**越大，分配就越可能失败。

这里值得强调的是，可以使用类似 *kmalloc* 函数中的 **priority** 参数调用 *get_free_pages* 和其他这些函数。某些情况下内存分配会失败，最经常的情形就是优先权为 **GFP_ATOMIC** 的时候。因此，调用这些函数的程序在分配出错时都应提供相应的的处理。

我们已经说过，如果不怕冒险的话，你可以假定按优先权 **GFP_KERNEL** 调用 *kmalloc* 和底层的 *get_free_pages* 函数都不会失败。一*般*说来这是对的，但有时也未必：我那台忠实可靠的 386，有着 4MB 的空闲的 RAM，但当我运行一个"play-it-dangerous"(冒险)模块时却象疯了一样。除非你有足够的内存，想写个程序玩玩，否则我建议你总检查检查调用分配函数的结果。

尽管 **kmalloc(GFP_KERNEL)**在没有空闲内存时有时会失败，但内核总是尽可能满足该内存分配请求。因此，如果分配太多内存，系统的响应性能很容易就会降下来。例如，如果往 *scull* 设备写入大量数据，计算机可能就会死掉；为满足 *kmalloc* 分配请求而换出内存页，系统就会变得很慢。所有资源都被贪婪的设备所吞噬，计算机很快就变的无法使用了；因为此时已经无法为你的 shell 生成新的进程了。我没有在 *scull* 模块中提到这个问题，因为它只是个例子模块，并不能真的在多用户系统中使用。但作为一个编程者，你必须要小心，因为模块是特权代码，会带来系统的安全漏洞(比如说，很可能会造成 DoS("denail-of-service")安全漏洞)。

使用一整页的 *scull*: *scullp*

至此，我们已经较完全地介绍了内存分配的原理，下面我会给出一些使用了页面分配技术的程序代码。*scullp* 是 *scull* 模块的一个变种，它只实现了一个裸(bare)设备－持久性的内存区域。和 *scull* 不同，*scullp* 使用页面分配技术来获取内存；scullp_order 变量缺省为 0，也可以在编译时或装载模块时指定。在 Linux 1.2 上编译的 *scullp* 设备在 order 大于零时会据绝被加载，原因我们前面已经说明过了。在 Linux 1.2 上，*scullp* 模块只允许“安全的”单页的分配函数。

尽管这是个实际的例子，但值的在这提到的只有两行代码，因为该设备其实只是分配和释放函数略加改动的 *scull* 设备。下面给出了分配和释放页面的代码行及其相关的上下文：

```
/*  此处分配一个单位内存  */
 if (!dptr->data[s_pos]) {
     dptr->data[s_pos] = (void *)__get_free_pages(GFP_KERNEL, dptr->order,0);
     if (!dptr->data[s_pos])
             return -ENOMEM;
     memset(dptr->data[s_pos], 0, PAGE_SIZE << dptr->order);

             /*  这段代码释放所有分配单元  */
             for (i = 0; i < qset; i++)
                 if (dptr->data[i])
                     free_pages((unsigned long)(dptr->data[i]), dptr->order);
```

从用户的角度看，可以感觉到的差异就是速度快了一些。我作了写测试，把 4M 字节的数据从 *scull0* 拷贝到 *scull1*,然后再从 *scullp0* 拷贝到 *scullp1*；结果表明内核空间处理器的使用率有所提高。

但性能提高的并不多，因为 *kmalloc* 设计得也运行得很快。基于页的分配策略的优点实际不在速度上，而是更有效地使用了内存。按页分配不会浪费内存空间，而用 *kmalloc* 函数则会浪费一定数量的内存。事实上，你可能会回想起第 5 章的“所使用的数据结构”一节中我们已经提到过 **select_table** 用了 *__get_free_page* 函数。

使用 *__get_free_page* 函数的最大优点是这些分配得到页面完全属于你，而且在理论上可以通过适当地调整页表将它们合并成一个线性区域。结果就允许用户进程对这些分配得到的不连续内存区域进行 *mmap*。我将在第 13 章的“mmap 设备驱动程序操作”一节中讨论 *mmap* 调用和页表的实现内幕。

vmalloc 和相关函数

下面要介绍的内存分配函数是 *vmalloc*，它分配*虚拟*地址空间的连续区域。尽管这段区域在物理上可能是不连续的(要访问其中的每个页面都必须独立地调用函数 *__get_free_page*)，内核却认为它们在地址上是连续的。分配的内存空间被映射进入内核数据段中，从用户空间是不可见的－这一点上与其他分配技术不同。*vmalloc* 发生错误时返回 0(**NULL** 地址)，成功时返回一个指向一个大小为 **size** 的线性地址空间的指针。

该函数及其相关函数的原型如下：

```
void* vmalloc(unsigned long size);
void vfree(void* addr);
void* vremap(unsigned long offset, unsigned long size);
```

注意在 2.1 版内核中 *vremap* 已经被重命名为 *ioremap*。而且，Linux 2.1 引入了一个新的头文件，**<linux/vmalloc.h>**，使用 *vmalloc* 时应将它包含进来。

与其他内存分配函数不同的是，*vmalloc* 返回很“高”的地址值－这些地址要高于物理内存的顶部。由于 *vmalloc* 对页表调整后允许用连续的“高”地址访问分配得到的页面，因此处理器是可以访问返回得到的内存区域的。内核能和其他地址一样地使用 *vmalloc* 返回的地址，但程序中用到的这个地址与地址总线上的地址并不相同。

用 *vmalloc* 分配得到的地址是不能在微处理器之外使用的，因为它们只有在处理器的分页单元之上才有意义。但驱动程序需要真正的物理地址时(象外设用以驱动系统总线的 DMA 地址)，你就不能使用 *vmalloc* 了。正确使用 *vmalloc* 函数的场合是为软件分配一大块连续的用于缓冲的内存区域。注意 *vmalloc* 的开销要比 *__get_free_pages* 大，因为它处理获取内存还要建立页表。因此，不值得用 *vmalloc* 函数只分配一页的内存空间。

使用 *vmalloc* 函数的一个例子函数是 *create_module* 系统调用，它利用 *vmalloc* 函数来获取被创建模块需要的内存空间。而在 *insmod* 调用重定位模块代码后，将会调用

memcpy_fromfs 函数把模块本身拷贝进分配而得的空间内。

用 *vmalloc* 分配得到的内存空间用 *vfree* 函数来释放，这就象要用 *kfree* 函数来释放 *kmalloc* 函数分配得到的内存空间。

和 *vmalloc* 一样，*vremap*(或 *ioremap*)也建立新的页表，但和 *vmalloc* 不同的是，*vremap* 实际上并不分配内存。*vremap* 的返回值是个虚拟地址，可以用来访问指定的物理内存区域；得到的这个虚拟地址最后要调用 *vfree* 来释放掉。

vremap 用于将高内存空间的 PCI 缓冲区映射到用户空间。例如，如果 VGA 设备的帧缓冲区被映射到地址 0xf0000000(典型的一个值)后，*vremap* 就可以建立正确的页表让处理机可以访问。而系统初始化时建立的页表只是用于访问低于物理地址空间的内存区域。系统的初始化过程并不检测 PCI 缓冲区，而是由各个驱动程序自己负责管理自己的缓冲区；PCI 的细节将在第 15 章“*外设总线概貌*”的“PCI 接口”一节中讨论。另外，你不必重映射低于 1MB 的 ISA 内存区域，因为这段内存空间可用其他方法访问，参见第 8 章“*硬件管理*”的“访问设备卡上的内存”一节。

如果你希望驱动程序能在不同的平台间移植，那么使用 *vremap* 时就要小心。在一些平台上是不能直接将 PCI 内存区域映射到处理机的地址空间的，例如 Alpha 上就不行。此时你就不能象普通内存区域那样地对重映射区域进行访问，你要用 *readb* 函数或者其他一些 I/O 函数(可参见第 8 章的“1M 内存空间之上的 ISA 内存”一节)。这套函数可以在不同平台间移植。

对 *vmalloc* 和 *vremap* 函数可分配的内存空间大小并没有什么限制，但为了能检测到程序员的犯下的一些错误，*vmalloc* 不允许分配超过物理内存大小的内存空间。但是记着，*vmalloc* 函数请求过多的内存空间会产生一些和调用 *kmalloc* 函数时相同的问题。

vremap 和 *vmalloc* 函数都是面向页的(它们都会修改页表)；因此分配或释放的内存空间实际上都会上调为最近的一个页边界。而且，*vremap* 函数并不考虑如何重映射不是页边界的物理地址。

vmalloc 函数的一个小缺点是它不能在中断时间内使用，因为它的内部实现调用了 **kmalloc(GFP_KERNEL)**来获取页表的存储空间。但这不是什么问题－如果*__get_free_page* 函数都还不能满足你的中断处理程序的话，那你还是先修改一下你的软件设计吧。

使用虚拟地址的 *scull*: *scullv*

使用了 *vmalloc* 的示例程序是 *scullv* 模块。正如 *scullp*，这个模块也是 *scull* 的一个变种，只是使用了不同的分配函数来获取设备用以储存数据的内存空间。

该模块每次分配 16 页的内存(在 Alpha 上是 128KB，x86 上是 64KB)。这里内存分配用了较大的数据块，目的是获取比 *scullp* 更好的性能，并且表明此时使用其他可行的分配技术相对来说会更耗时。用*__get_free_pages* 函数来分配一页以上的内存空间容易出错，而且即使成功了，也相对较慢。前面我们已经看到，用 *vmalloc* 分配若干页比其他函数要快一些，

但由于存在建立页表的开销，只分配一页时却会慢一些。*scullv* 设计得和 *scullp* 很相似。**order** 参数指定分配的内存空间的“幂”，缺省为 4。*scullv* 和 *scullp* 的唯一差别在下面一段代码：

```
/*  此处用虚拟地址来分配一个单位内存  */
 if (!dptr->data[s_pos]) {
     dptr->data[s_pos] = (void *)vmalloc(PAGE_SIZE <<order);
     if (!dptr->data[s_pos])
             return -ENOMEM;

             /*  这段代码释放所有分配单元  */
             for (i = 0; i < qset; i++)
                 if (dptr->data[i])
                     vfree (dptr->data[i]);
```

如果你在编译这两个模块时都打开了调试开关，就可以通过读它们在*/proc* 下创建的文件来查看它们进行的数据分配。下面的快照取自我的计算机，我机器的物理地址是从 0 到 0x1800000(共 24MB)：

```
morgana.root# cp /bin/cp /dev/scullp0
morgana.root# cat /proc/scullpmem
Device 0: qset 500, order 0, sz 19652
   Item at 0063e598, qset at 006eb018
        0: 150e000
        1:   de6000
        2: 10ca000
        3:   e19000
        4:   bd1000

morgana.root# cp /zImage.last /dev/scullv0
morgana.root# cat /proc/scullvmem
Device 0: qset 500, order 4, sz 289840
   Item at 0063ec98, qset at 00b3e810
        0: 2034000
        1: 2045000
        2: 2056000
        3: 2067000
        4: 2078000
```

从这些值可以看到，*scullp* 分配物理地址(小于 0x1800000)，而 *scullv* 分配虚拟地址(但注意实际数值与 Linux 2.1 会不同，因为虚拟地址空间的组织形式变了－见第 17 章“*近期发展*”的“虚拟内存”一节)。

“脏”的处理方法(Playing Dirty)

如果你确实需要大量的连续的内存用作缓冲区，最简单的(也是最不灵活的，但也最容易出错的)方法是在系统启动时分配。显然，模块不能在启动时分配内存；只有直接连到内核的设备驱动程序才能运行这种“脏”的处理方式，在启动时分配内存。

尽管在启动时就进行分配似乎是获得大量内存缓冲区的唯一方法，但我还会在第 13 章的“分配 DMA 缓冲区”一节中介绍到另一种分配技术(虽然可能更不好)。在启动时分配缓冲区有点“脏”，因为它跳过了内核内存管理机制。而且，这种技术普通用户无法使用，因为它要修改内核。绝大多数用户还是愿意装载模块，而并不愿意对内核打补定或重新编译内核。尽管我不推荐你使用这种“分配技术”，但它还是值得在此提及的，因为在 **GFP_DMA** 被引入之前，这种技术曾是 Linux 的早期版本里分配可用于 DAM 传输的缓冲区的唯一方法。

让我们先看看启动是是如何进行分配的。内核启动时，它可以访问系统所有的内存空间。然后以空闲内存区域的边界作为参数，调用内核的各个子系统的初始化函数进行初始化。每个初始化函数都可以“偷取”一部分空闲区域，并返回新的空闲内存下界。由于驱动程序是在系统启动时进行内存分配的，所以可以从空闲 RAM 的线性数组获取连续的内存空间。

除了不能释放得到的缓冲区，这种内存分配技术还有些缺点。驱动程序得到这些内存页后，就无法将它们再放到空闲页面池中了；页面池是在已经物理内存的分配结束后才建立起来的，而且我也不推荐象这样“黑客”内存管理的内部数据结构。但另一方面，这种技术的优势是，它可以获取用于 DMA 传输等用途的一段连续区域。目前这也是分配超过 32 页的连续内存缓冲区的唯一的“安全”的方式，32 页这个值是源于 *get_free_pages* 函数参数 order 可取的最大值为 5。但是如果你需要的多个内存页可以是物理上不连续的，最好还是用 *vmalloc* 函数。

如果你真要在启动时获取内存的话，你必须修改内核代码中的 *init/main.c* 文件。关于 *main.c* 文件的更多细节可参见第 16 章和第 8 章的“1M 内存空间之上的 ISA 内存”一节。

注意，这种“分配”只能是按页面大小的倍数进行，而页面数不必是 2 的某个幂次。

快速参考

与内存分配有关的函数和符号列在下面：

#include <linux/malloc.h>
void *kmalloc(unsigned int size, int priority);
void kfree(void *obj);
这两个函数是最常用的内存分配函数。

#include <linux/mm.h>
GFP_KERNEL
GFP_ATOMIC

GFP_DMA

kmalloc 函数的优先权。**GFP_DMA** 是个标志位，可以与 **GFP_KERNEL** 和/或 **GFP_ATOMIC** 相或。

unsigned long get_free_page(int priority);
unsigned long __get_free_page(int priority);
unsigned long __get_dma_pages(int priority, unsigned long order);
unsigned long __get_free_pages(int priority, unsigned long order,int dma);

这些都是面向页的内存分配函数。以下划线开头的函数不清零分配而得的页。只有前两个函数是在 1.2 版和 2.0 版的 Linux 间可移植的，而后两者在 1.2 版与 2.0 版中的行为并不同。

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

这些函数用于释放面向页的分配得到的内存空间。

void* vmalloc(unsigned long size);
void* vremap(unsigned long offset, unsigned long size);
void vfree(void* addr);

这些函数分配或释放连续的虚拟地址空间。*vremap* 用*虚拟*地址访问物理内存(在 2.1 版的 Linux 中被称为 *ioremap*)，而 *vmalloc* 是用来分配空闲页面。两种情况下，都是用 *vfree* 来释放分配的内存页。2.1 版的 Linux 引入了头文件**<linux/vmalloc.h>**，使用这些函数时必须先包含(#include)这个头文件。