互动出版网 China-Pub.com
专业图书网上第一专营店
www.china-pub.com 独家提供样章

第5章 内存的分配和释放

- 变量
- 动态内存
- 动态内存实例
- 内存池
- 内存池的实例
- 内存的分配和释放函数

要是问到经验丰富的程序员程序中最重要的要素是什么？他们一定会回答是数据的结构。创建程序时数据的结构如果不合理，不仅难以维护，而且程序的稳定性也是很关键的。设备驱动程序的内部包含很多数据结构，处理该数据时，必须使用变量和动态内存。

作为内核的一部分运行的设备驱动程序，其变量和动态内存与普通的应用程序有所区别。要分别掌握局部变量或全局变量的定义，还要熟悉用于运行I/O类型的定义方式。为了创建多平台上可移植的设备驱动程序，应尽可能使用内核定义的数据类型。另外，设备驱动程序中使用的动态内存分配/释放函数及其处理模式与创建应用程序时使用的内存分配/释放函数的形式和功能完全不同。设备驱动程序具有在多进程环境或DMA等需要具备特殊的物理条件的内存分配和中断状态下还能稳定分配内存的函数。

本章介绍了作为编程基础的变量使用方法，及动态内存和内存共享相关的内容。通过本章的各种实例，可以掌握具体的使用方法。本章的基本目的在于熟悉下一章节中可能涉及到的内容，并且掌握常用的内存分配函数，因此实例的难度较低。

- 2.4内核和2.6内核共同使用的函数

  kmalloc ( ), kfree ( ), vmalloc ( ), vfree ( ) , _ _ get_free_pages ( ), free_ pages ( )

- 2.6内核中新支持的函数

  mempool_create ( ), mempool_destroy ( ), mempool_alloc ( ), mempool_free ( )

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

5.1　变量

设备驱动程序中的变量对可移植性和多进程环境及内核的稳定性都具有重要意义，因此，我们需要先掌握C语言变量的特性，并理解Linux内核中的应用方法。本节主要介绍了创建局部变量和全局变量的方法，函数命名与变量命名的方法，以及控制变量确保可移植性的方法。最后，还介绍了用于控制I/O内存的gcc特性及相关内容。

5.1.1　局部变量和全局变量的选择

函数内的局部变量和函数外的全局变量，将在编译的起点定义大小和地址（对于模块，在模块代码上指定相对位置）。因此，设备驱动程序源代码中定义的局部变量和全局变量都分配在内核的内存空间上。

要想控制设备驱动程序，就要为变量指定局部或者全局，这与程序设计员的编程习惯有关系。但是设备驱动程序在多进程环境中运行时，为了使各个进程合理调用变量，最好使用可行的局部变量。在内核上加载或删除某个设备驱动程序时，必须指定全局变量。

5.1.2　防止函数和变量的重复命名

Linux内核源代码TEXT超出了50MB，其中包含了大量的函数和全局变量，因此，新增的设备驱动程序源代码中可能出现重复的函数和变量。每一个符号表（symbol table）对应一个模块，对于模块而言，重复命名而发生错误的几率很小，但在包含内核的源代码时，有可能发生严重的错误。为了防止发生此类错误，最好在函数或变量上使用static关键字。当然，这并不是强行要求。为设备驱动程序的函数使用特殊的前缀可以减少出现重复的函数名，但是仍有可能发生错误。

定义函数和变量的过程中使用static关键字后，只有对应的文件源代码才会引用，因此，在链接的过程中不要把函数名或变量名直接引入到符号表上。

其他设备驱动程序定义外部引用时不能使用static关键字。另外，局部变量上表示static的意义有所不同，不能在局部变量上使用该关键字。若在局部变量上表示了static，结果与全局变量相同，终止了函数后仍能保持相应的值。局部变量与全局变量的区别在于不能引用外部符号表。通常，以下列方式表示函数或变量。

```
static int checkout  =  0;     //   变量实例
static int dev_app_check (void)  //   函数实例
{
    ⋮
}
```

5.1.3　可移植性和数据类型

Linux内核是在多个进程中运行的多平台内核。目前，微处理器正趋于从32位升至62位，因此要更关注平台之间的互换性。为了确保多个平台上的可移植性，除了考虑平台之间的控制方式，还要明确变量的数据类型。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

1. 变量的数据类型

变量的数据类型中要注意int类型。C语言编程时，多数数据变量使用int，但是int型变量的大小依赖于进程。16位微处理器中int型变量的大小为2字节，但是在32位微处理器中是32位，64位微处理器中是64位。同样地，long型的大小也随微处理器的类型而发生变化。变量类型的大小差异会对其他平台之间的移植性带来负面影响。为了解决此类问题，最好使用Linux内核提供的数据类型。下列头文件表示了该数据类型。

```
# include <asm/ types.h>
```

表5-1归纳了常用的数据类型。

表5-1 变量的数据类型

带符号的正数		不带符号的正数	
__s8，s8	8位（字节）	__u8，u8	8位（字节）
__s16，s16	16位（字）	__u16，u16	16位（字）
__s32，s32	32位	__u32，u32	32位
__s64，s64	64位	__u64，u64	64位

应用程序中可以使用带两条下划线的变量类型，但是并不是说所有设备驱动程序都使用这些数据类型。本书的例子中也没有使用上述数据类型。由于作者的目的在于展现可行、熟悉的代码形式，因此考虑到可移植性，最好使用内核提供的数据类型。

2. 结构体

C语言中的结构体与编译器有关，通常表示为int型大小的倍数。一般该特性不会引起太大的错误。然而，表现controller 的register或与应用程序交换数据时，若表示为int型的倍数，就会经常发生错误，此时，如下列内容，可使用packed关键字定义实际变量的大小。

```
typedef struct
{
      u16 index;
      u16 data;
      u8 data2
}__ attribute__ ((packed)) testctl_t;
```

3. 字节顺序

移植到其他平台时应注意正数字节的顺序。存储正数数据时，不同的微处理器在内存中存放最小字节的位置有所不同。先存储最小字节时称为低字节（little endian），相反，最后存储最小字节时称为高字节（big endian），如图5-1所示。

下面这些头文件都在include/linux/byteorder/目录中。

```
# include < linux/byteorder/big_endian.h>      : 高字节处理
# include < linux/byteorder/little_endian.h>   : 低字节处理
# include < linux/byteorder/pdp_endian.h>      : PDP字节处理
# include < linux/byteorder/generic.h>         : 定义互换性
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
# include < linux/byteorder/swab.h>            : 转换字节的宏
# include < linux/byteorder/swabb.h>           : 转换字节的宏
```

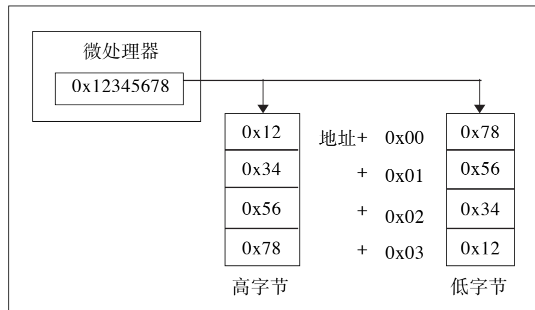

图5-1　按字节顺序排列的高字节和低字节

实际上设备驱动程序并没有包含上述所有的头文件，而是包含了下列头文件。

```
# include < asm / byteorder.h >
```

包含该头文件后，能够明确内核源代码使用的字节方式。通常，使用低字节方式时定义为_ _LITTLE_ENDIAN，使用高字节方式时定义为_ _BIG_ENDIAN。

4. 数据的表示

除了i386系列系统外，请求有效处理数据或表示数据时，均使用运行其他处理的微处理器。例如，对于32位进程，以4个字节为单位表示数据。通常，在编译器的链接脚本中指定表示方式，对于动态的数据，可以获取内核内部宏函数的帮助。

```
# include <asm/ unaligned.h>
get_unaligned (ptr)      : ptr地址中读取值
put_unaligned (val, ptr): ptr地址中写入val值
```

实际上，在网络设备或IDE设备上能够看到使用这两个宏的设备驱动程序。

5.1.4　输入输出内存变量的处理

部分微处理器支持专用的输入输出处理命令。当微处理器配置了输入输出处理命令时，可以直接调用相应的命令，对于使用内存地址的输入输出，一定要特别注意该处理过程。

另外，对于以PCI方式成为主流的设备，多数采用内存地址的处理方式。例如，为了在0xE0000300地址上读取或写入数据，执行下列命令。

```
u32 *ptr= (u32 *) 0xE0000300;

*ptr=0x1234;
*ptr=*ptr &0xFF;
```

通常情况下不会发生错误，但是优化编译选项后，也可能会发生错误。从编译器的角度查看上述内容，可在0x1234上代入0xFF，直接运行下列语句。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
*ptr=0x1234 &0xFF;
```

但是控制硬件时，优化后的结果往往不能使设备正常运行，此时可以参考下例，使用volatile重新指定编译内容。

```
volatile  u32 *ptr= (volatile  u32 * ) 0xE0000300;
```

如果将变量进行volatile修饰，则编译器对此变量的读写操作都不会被优化（肯定执行），也就是说编译器不会对用这个关键字修饰的变量进行优化，每次都通过本变量的内存地址读取数据。

5.2 动态内存

由于是设备驱动程序，必须管理动态内存，仅使用静态变量或stack变量不能创建程序。普通程序可以简单地分配和释放动态内存，作为内核的一部分而运行的设备驱动程序必须使用内核提供的函数，因此，动态内存的分配和释放方式与应用程序完全不同。本节重点介绍了内核中执行的内存函数的类型、各个函数的特点及使用方式。

偶尔，设计者为了达到特殊目的会使用变量和队列进行编程，但是处理大小可变的信息，或处理List等结构的数据时，必须动态分配内存。使用完内存，必须释放内存，此时最常用的函数是malloc ( )。如下所示，该函数从系统中分配到一定内存，返回其起始地址，运行结束后再使用free ( )函数释放内存到系统。

```
# include <stdlib.h>
char *buff;
buff=malloc(1024);
if (buff== NULL) exit (1);
      ⋮
free (buff);
```

设备驱动程序不能使用C语言标准函数库中的malloc( )和free( )函数。由于下列特性和情况，不能仅以简单的结构实现内存分配和释放。内核中对内存的管理是非常复杂的，下面列出了内核中内存管理的特殊性。

（1）进程的用户内存空间和其他内核内存空间的特性

对于进程运行中的用户端内存空间，由内核的内存管理程序和glibc的内核分配程序分配内存，因此，不会发生内存分配和释放的错误，但是，设备驱动程序运行的内存空间是内核的内存空间。特别地，内核可以映射系统整体的物理内存，并且自行管理，因此需要考虑MMU（内存管理单元）。

内核或设备驱动程序运行的内存区域主要在内核的内存，并且分配和释放机制又与进程的内存分配和释放方式不同。希望了解内核分配详细内容的读者，可以先阅读《UNIX内部结构》（U.Vahalia著）或“The Design and Implementation of 4.4 BSD”中的参考文献。历史上是使用了多种类型的内核内存分配器，并且直到初期都没有定义在内核源代码的内部。

也就是说，内核和进程的内存空间是互不相关的不同结构。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

（2）以PAGE_SIZE为单位分配内存的特性

系统会随时分配内存，重复分配和释放了内存时会发生内存分段现象，因此，利用特定的单位分配内存。Linux中利用PAGE_SIZE和PAGE_SHIFT进行管理。特别地，PAGE_SIZE值受MMU管理单元的影响。通常，PAGE_SIZE值分配为4KB。

（3）系统中不存在虚拟内存

应用程序中利用malloc ( )函数分配内存时，由虚拟内存进行处理，因此失败的可能性很小。但是设备驱动程序中分配内存时，会发生内存不足的现象。通常，内核内存不使用虚拟内存的内存分页功能，因此需要合并或移动现存的内存，从而确保实际可利用的内存大小。此时，设备驱动程序的特性决定等待内存分配，还是返回值。

（4）虚拟内存的映射内存在辅助设备的情况

设备驱动程序映射的内存已被分配，但是受内核中运行的虚拟内存处理方式的支配，内存交换到辅助记忆设备中，此时我们要充分考虑相应的处理方式。

（5）DMA等设备中使用的连续物理内存地址的情况

设备驱动程序控制的设备利用DMA高速传送数据。DMA需要连续的物理内存，此时需要相应的内存管理例程。目前使用虚拟DMA的设备不是很多。

（6）中断状态下分配内存

中断（interrupt）状态不能维持到内核中内存不足而重新分配内存的过程。另外，为了迅速分配内存，需要具有内存管理例程。

内核分配内存时为了满足上述特性及状况，提供了下列多种内存分割函数。

```
kmalloc( ) , kfree ( )
__ get _free_pages ( ), free _pages ( )
vmalloc( ), vfree ( )
```

5.2.1　函数kmalloc( )和kfree ( )

这两个函数不仅分配速度快，使用方法也简单，是设备驱动程序中使用最多的函数。使用方法与malloc ( )，free ( )函数相似，具体参考下列内容。因此，只要了解分配函数的特性就不难应用此类函数。

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

char *buff;

buff = kmalloc (1024, GEP_KERNEL);
if (buff !=NULL)
{
     ⋮
     kfree (buff);
}
else
{
     printk ("kmalloc error\n");
}
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

使用kmalloc ( )函数时可分配的大小为32 × PAGE_SIZE。大多数系统中PAGE_SIZE的大小为4096，因此超过131072字节就不能分配内存。另外，与malloc ( )函数的区别在于为分配的内存指定特性或在内存分配的处理方式上给定下列变量值。

（1）GFP_KERNEL

kmalloc ( )函数中使用的代表性参数，请求动态内存总是分配成功。利用该值使用kmalloc ( )函数时，若内核中的内存不足（剩余内存小于min_free_pages），设备驱动程序呼叫的进程将被停止运行，直到动态内存可以分配。当其他进程释放内存，达到可分配动态内存时，才能使程序重新启动。kmalloc ( )函数分配到必要的内存，并返回相应内存的起始地址，因此，在中断服务上使用kmalloc ( )函数时不能利用该参数。

（2）GFP_ATOMIC

内核存在可分配的内存时，无条件分配内存，若没有就立即释放NULL，因此，进程不会进入睡眠状态，但是编程时有可能无法分配内存。

（3）GFP_DMA

用于分配连续的物理内存。设备驱动程序运行的内存空间不是物理内存而是虚拟地址内存，从进程的角度查看分配的内存，虚拟地址空间是连续的，而实际物理空间是不连续的。DMA控制器无法使用不连续的物理空间，此时就可以使用GFP_DMA标志。

5.2.2 函数vmalloc ( )和vfree ( )

如下列内容所示，vmalloc ( )函数除了分配大小外，不设置变量，因此与malloc ( )函数非常相似。

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


char *buff;


buff = vmalloc (1024);
if (buff !=NULL)
{
     ⋮
     vfree (buff);
}
else
{
     printk  ("vmalloc error\n");
}
```

kmalloc( )函数限定了分配大小，只要虚拟空间允许vmalloc ( )函数就不受分配大小的限制，因此，只用于分配内存的大空间。

在设备驱动程序中无论使用vmalloc ( )函数的分配地址，还是kmalloc ( )函数中分配到的地址都没有太大的差异。如果要得到相应地址的实际（物理）地址，由于vmalloc ( )函数分配虚拟地址空间，相应地址的区域可能在硬盘上，因此有可能失败。另外，vmalloc ( )函数

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

为了分配大容量连续空间执行虚拟内存的管理命令，与kmalloc ( )函数相比分配速度较慢。分配内存时kmalloc ( )函数给出的是进程不能进入睡眠的标志，因此不会有错误；而vmalloc ( )函数没有相应设置，中断服务函数中不能调用该函数。

5.2.3 函数__get_free_pages( )和free_pages( )

kmalloc ( )函数和vmalloc ( )函数分配指定大小的内存空间。可以以PAGE_SIZE为单位分配内存，以PAGE_SIZE单位相除时不能分配到更大的空间。除了该分配方法，Linux内核还提供_ _get_free_pages ( )函数类型的内存页为单元的分配函数。分配函数的大小，单位为PAGE_SIZE。实际工作中能很好地调用该函数，在这里省略了详细地说明，简单列出了基本的使用方法。经常使用到的函数如下：

```
unsigned long FASTCALL(_ _get_free_pages (unsigned int gfp_mask, unsigned int
                        order));
void FASTCALL(free_pages(unsigned long addr, unsigned int order));
```

另外，利用上述函数的宏函数时，只要理解上述两个函数就行。在不同的内核版本中，上述两个函数分别包含在下列头文件中。

- 2.4内核 #include <linux/mm.h>
- 2.6内核 #include<linux/gfp.h>

_ _get_free_pages ( )函数的第一个变量gfp_mask可以直接使用kmalloc( )函数中使用的参数，但是，第二个变量不是指定变量的大小，而是指定次数，该值为PAGE_SIZE的两倍。因为大小计算比较复杂，Linux内核中提供了称为get_order ( )的宏函数。为了使用get_order ( )宏，必须包含#include<asm/page.h>。使用_ _get_free_pages ( )的实例如下。

```
char *buff;
int order;


order = get_order (8192);
buff = _ _get_free_pages (GEP_KERNEL, order);
if (buff !=NULL)
{
        ⋮
    free_pages (buff, order);
}
```

使用该函数时一定要注意可使用的最大值定义为order值。该最大值定义为MAX_ORDER，其值通常为11。次数过高分配失败的几率也高，通常使用小于5的值，即只分配32 × PAGE_SIZE大小的内存。该值与kmalloc( )函数可分配最大值一致。

5.3 动态内存实例

本节中介绍了kmalloc ( )、kfree( )、vmalloc ( )、vfree ( )、_ _get_free_pages ( )等内存

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

分配函数的应用实例。实例非常简单，没有必要解释源代码，分别介绍了2.4内核和2.6内核中的使用方法。对比两个实例就能发现，结构上没有太大的区别。

5.3.1 实例源代码

[实例5-1] 2.4内核 (2.4/basicmem/basicmem.c)

```
01 #define MODULE
02 #include <linux/module.h>
03 #include <linux/kernel.h>
04
05 #include <linux/slab.h>
06 #include <linux/vmalloc.h>
07 #include <asm/page.h>
08
09 void kmalloc_test (void)
10 {
11     char *buff;
12
13     printk ("kmalloc test\n");
14
15     buff = kmalloc (1024, GEP_KERNEL);
16     if (buff !=NULL)
17     {
18          sprintf (buff,"Small Memory Ok\n");
19          printk (buff);
20
21          kfree (buff);
22     }
23
24     buff = kmalloc (32 * PAGE_SIZE, GEP_KERNEL);
25     if (buff !=NULL)
26     {
27          printk ("Big Memory Ok\n");
28          kfree (buff);
29     }
30
31 }
32
33 void vmalloc_test (void)
34 {
35     char *buff;
36
37     printk ("vmalloc test\n");
38
39     buff = vmalloc (33 * PAGE_SIZE);
40     if (buff !=NULL)
41     {
42          sprintf (buff, "vmalloc test ok\n");
43          printk (buff);
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
44
45          vfree (buff);
46      }
47
48 }
49
50 void get_free_pages_test (void)
51 {
52      char *buff;
53      int order;
54
55      printk ("get_free_pages test\n");
56
57      order = get_order (8192*10);
58      buff = __get_free_pages (GFP_KERNEL, order);
59      if (buff !=NULL)
60      {
61          sprintf(buff,"__get_free_pages test ok [%d]\n", order);
62          printk (buff);
63
64          free_pages (buff, order);
65      }
66 }
67
68 int init_module (void)
69 {
70      char *data;
71
72      printk ("Module Memory Test\n");
73
74      kmalloc_test ( );
75      vmalloc_test ( );
76      get_free_pages_test ( );
77
78      return 0;
79 }
80
81 void cleanup_module (void)
82 {
83      printk ("Module Memory Test End\n");
84 }
```

[实例5-2]　2.6内核 (2.6/basicmem/basicmem.c)

```
01 #include <linux/init.h>
02 #include <linux/module.h>
03 #include <linux/kernel.h>
04
05 #include <linux/slab.h>
06 #include <linux/vmalloc.h>
07
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
08 void kmalloc_test (void)
09 {
10     char *buff;
11
12     printk ("kmalloc test\n");
13
14     buff = kmalloc (1024, GEP_KERNEL);
15     if (buff !=NULL)
16     {
17          sprintf (buff, "test memory\n");
18          printk (buff);
19
20          kfree (buff);
21     }
22
23     buff = kmalloc (32 * PAGE_SIZE, GEP_KERNEL);
24     if (buff !=NULL)
25     {
26          printk ("Big Memory Ok\n");
27          kfree (buff);
28     }
29
30 }
31
32 void vmalloc_test (void)
33 {
34     char *buff;
35
36     printk ("vmalloc test\n");
37
38     buff = vmalloc (33 * PAGE_SIZE);
39     if (buff !=NULL)
40     {
41          sprintf (buff, "vmalloc test ok\n");
42          printk (buff);
43
44          vfree (buff);
45     }
46
47 }
48
49 void get_free_pages_test (void)
50 {
51     char *buff;
52     int order;
53
54     printk ("get_free_pages test\n");
55
56     order = get_order (8192*10);
57     buff = __get_free_pages (GFP_KERNEL, order);
58     if (buff !=NULL)
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
59      {
60           sprintf (buff,"__get_free_pages test ok [%d]\n", order);
61           printk (buff);
62
63           free_pages (buff, order);
64      }
65 }
66
67 int memtest_init (void)
68 {
69      char *data;
70
71      printk ("Module Memory Test\n");
72
73      kmalloc_test ( );
74      vmalloc_test ( );
75      get_free_pages_test ( );
76
77      return 0;
78 }
79
80 void memtest_exit (void)
81 {
82      printk ("Module Memory Test End\n");
83 }
84
85 module_init (memtest_init);
86 module_exit (memtest_exit);
87
88 MODULE_LICENSE ("Dual BSD/GPL");
```

5.3.2　运行方法

编译实例，并利用insmod命令将生成的模块插入内核中，dmesg中显示如下内容。

```
[2.4内核]
    Module Memory Test
    kmalloc test
    Small Memory Ok
    Big Memory Ok
    vmalloc test
    vmalloc test ok
    get_free_pages test
    __get_free_pages test ok [5]

[2.6内核]
    Module Memory Test
    kmalloc test
    test memory
    Big Memory Ok
    vmalloc test
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
vmalloc test ok
get_free_pages test
__get_free_pages test ok [5]
```

最后一行“_ _get_free_pages test ok [5]”的数字是内存分配次数。测试阶段分配的内存空间过大时，发生segment fault错误，无法停止模块，此时只能重新启动系统。

5.4 内存池

本节中介绍2.6内核新增概念内存池（memory pool）。其使用方法和概念都较为简单，所有读者都可以轻松地学会其中内容。另外，即使不去理解这一节的内容，设备驱动程序的编程也不会受到太大的影响，因此视情况可以跳过本节内容。

2.6内核中提供了所谓内存池（memory pool）的新的内存管理方法。前缀为“mempool_”的内存池方式不能直接管理系统内存，并分配和释放内存。已有的kmalloc、vmalloc等函数足以实现该功能。那么，2.6内核中增加内存池方式的原因，以及具体作用是什么？

2.6内核中为了适应企业环境进行了部分改动，因此，内核开发的主要目的在于改善处理速度，并且提高数据处理量，但是，最大的问题在于内存。在处理大量的数据时，系统内核内部的可使用内存量就会瞬间减少甚至不足。通常，此时启动虚拟文件系统，它的处理速度慢，并且只对进程可分配的内存适用。为使系统运行更佳自如，需注意不属于进程范围的内存分配，即内核的企业服务函数内部或设备驱动程序函数请求的内存分配，因此，2.6内核预先分配了预测的最小尺寸内存，并体现了新的管理方式内存池的概念。内存池主要用在块设备驱动程序上。

内存池由API构成，它具有动态内存的分配和释放功能。

- 管理员的生成和取消API

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

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

- 分配和释放API

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

使用方法不是很复杂。首先使用mempool_create ( )函数生成内存管理结构体，此时把预先分配的数量指定为min_nr。此外，实际分配方式中，以alloc_fn通过指定用户选择的分配函数生成内存对象，并且以free_fn通过指定用户选择的释放函数释放分配的内存对象。

为了实际分配到内存，使用mempool_alloc ( )函数。如果内存池分配器无法提供内存，那么就可以用预分配的池。若采用用户所选择的分配函数alloc_fn分配内存失败，则调用mempool_alloc ( )函数就可以获取预分配的内存对象，因此，mempool_alloc ( )函数一定会

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

成功分配到内存，这样可以防止进程在分配内存失败后进入睡眠状态。当然，使用完预分配内存后，再也没有可分配的内存，此时由向mempool_alloc ( )函数传达的gfp_mask分配标记，可能分配失败或者进程进入睡眠，因此，应在考虑设备驱动程序的内存使用量后，再写入mempool_create ( )函数指定的min_nr值。

内存池API的内存分配函数不指定大小，即用户可以定义分配函数，指定内存的大小。mempool_alloc ( )函数的缺点在于调用后只能分配到指定大小的内存，而优点在于调用mempool_alloc ( )函数时用户分配函数可以初始化内存。

5.5　内存池的实例

本节介绍了利用内存池API分配和释放内存的实例。通过本章的学习可以利用printk方式直接确认用户定义的内存分配函数的调用起点和调用次数，以及释放函数的调用起点及调用次数。

5.5.1　实例源代码

1. 实例

[实例5-3]　利用内存池API的内存分配和释放(2.6/mempool/mepool.c)

```
01 #include <linux/init.h>
02 #include <linux/module.h>
03 #include <linux/kernel.h>
04 #include <linux/slab.h>
05 #include <linux/mempool.h>
06
07 #define MIN_ELEMENT     4
08 #define TEST_ELEMENT    4
09
10 typedef struct
11 {
12     int number;
13     char string[128];
14 } TMemElement;
15
16
17 int elementcount = 0;
18
19 void *mempool_alloc_test (int gfp_mask, void *pool_data)
20 }
21     TMemElement *data;
22     printk ("----> mempool_alloc_test\n");
23
24     data = kmalloc (sizeof (TMemElement), gfp_mask);
25     if (data ! =NULL) data -> number =elementcount++;
26     return data;
27 }
28
29 void mempool_free_test (void *element, void *pool_data)
```

互动出版网
China-Pub.com
专业图书网上第一专营店
www.china-pub.com
独家提供样章


```
30 {
31      printk ("----> call mempool_free_test\n");
32      if (element ! =NULL) kfree (element);
33 }
34 int mempool_init (void)
35{
36      mempool_t    *mp;
37      TMemElement *element[ TEST_ELEMENT];
38      int         lp;
39
40      printk ("Module MEMPOOL Test\n");
41
42      memset (element, 0, sizeof (element));
43
44      printk ("call mempool_create\n");
45      mp = mempool_create (MIN_ELEMENT, mempool_alloc_test,
        mempool_free_test, NULL);
46
47      printk ("mempool allocate\n");
48      for (lp = 0; lp < TEST_ELEMENT; lp++)
49      {
50           element [lp] = mempool_alloc (mp, GFP_KERNEL);
51           if (element [lp] = = NULL) printk ("allocate fail\n");
52           else
53           {
54                     sprintf (element [lp] -> string, "alloc data %d\n",
                       element [lp] -> number);
55                     printk (element [lp] -> string);
56           }
57      }
58
59      printk ("mempool free\n");
60      for (lp = 0; lp < TEST_ELEMENT; lp++)
61      {
62           if (element [lp] ! = NULL) mempool_free (element [lp], mp);
63      }
64
65      printk ("call mempool_destroy\n");
66      mempool_destroy (mp);
67
68          return 0;
69      }
70
71      void mempool_exit (void)
72
73          printk ("Module MEMPOOL Test End\n");
74      }
75
76      module_init (mempool_init);
77      module_exit (mempool_exit);
78      MODULE_LICENSE ("Dual BSD/GPL");
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

2. 源代码解析

（1）[mempool.c] 19~27行

mempool_alloc_test ( )通过kmalloc( )函数分配TMemElement结构体大小的内存。在分配到的结构体number域上记录分配次数。

（2）[mempool.c] 29~33行

由mempool_free ( )函数调用mempool_free_test。该函数再次释放已分配的内存。

（3）[mempool.c] 34行

mempool_init ( )函数将记录最先从内存池分配到的对象地址的element队列初始化为0。element队列用来记录地址，应写入NULL。

（4）[mempool.c]44~45行

以printk方式表示调用mempool_create( )函数，并调用mempool_create( )函数登记内存分配函数mempool_alloc_test( )和内存释放函数mempool_free_test( )。此时预分配的内存对象为MIN_ELEMENT。成功分配内存后，mp结构体内部保存了内存池管理者结构体地址。内存池分配和释放函数中返回分配地址mp。

（5）[mempool.c]47~57行

以printk方式输出“mempool allocate”字符，表示开始分配内存。利用mempool_alloc ( )函数，以TEST_ELEMENT大小分配内存，此时调用mempool_alloc_test( )函数。利用sprintf( )函数缓存中写入分配次数域number值，再利用printk( )函数输出缓存中的内容。

（6）[mempool.c]59~63行

以printk方式输出“mempool free”字符，表示释放已分配的内存。释放内存后，以TEST_ELEMENT大小运行mempool free ( )函数。

（7）[mempool.c]65~66行

以printk方式输出“call mempool_destroy\n”字符，表示回收内存结构体。此外，调用mempool_destroy ( )清除内存管理员。

（8）[mempool.c]71~74行

利用rmmod命令从内核中移除已加载的模块后，调用mempool_exit( )函数。该函数不执行特殊的任务。

（9）[mempool.c]76行

编译mempool.c生成mempool.ko模块。利用insmod实用程序把该模块插入到内核，从而调用mempool_init( )函数。

5.5.2　运行方法

插入该模块后，利用dmesg命令查看内核信息的结果如下。

```
Module MEMPOOL Test
call mempool_create
----> mempool_alloc_test
----> mempool_alloc_test
----> mempool_alloc_test
----> mempool_alloc_test
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
mempool allocate
----> mempool_alloc_test
alloc data 4
----> mempool_alloc_test
alloc data 5
----> mempool_alloc_test
alloc data 6
----> mempool_alloc_test
alloc data 7
mempool free
----> call mempool_free_test
----> call mempool_free_test
----> call mempool_free_test
----> call mempool_free_test
call mempool_destroy
----> call mempool_free_test
----> call mempool_free_test
----> call mempool_free_test
----> call mempool_free_test
Module MEMPOOL Test End
```

查看运行结果，调用mempool_create ( )时，以MIN_ELEMENT调出mempool_alloc_test ( )函数，然后再以mempool_alloc ( )函数的大小调出mempool_alloc_test( )函数。此时分配的序号从MIN_ELEMENT所表示的序号开始递增（该实例从4号开始）。

调用mempool_free ( )函数时，随之调出mempool_free_test( )函数。调用mempool_destroy ( )函数后，再调出mempool_free_test( )函数释放已分配但为运行分配处理的内存。

内存池函数以预分配方式解决系统内存不足时的问题。

5.6　内存的分配与释放函数

Kmalloc ( ) 函数

功能：	分配内存。
原型：	#include <linux/slab.h> Void *kmalloc (size_t size, int flag);
说明：	分配内核内部的动态内存。内存分配速度快，但是大小限制在32×PAGE_SIZE以内。另外，分配内存后，不能释放内存，因此分配内存后，重新分配回收的内存时仍保留原值。
变量：	• flag变量　指定将分配内存的特性； • GFP_KERNEL　求常见的内存分配； • GFP_ATOMIC　请求立即分配； • GFP_DMA　请求连续的物理内存空间。
返回值：	成功返回分配内存的起始地址，失败返回NULL。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

Kfree ( ) 函数

功能：	释放分配的内存。
原型：	#include <linux/slab.h> Void kfree (const void *addr)；
说明：	释放kmalloc分配的动态内存。
变量：	• addr　分配的动态内存地址。

Vmalloc ( ) 函数

功能：	分配动态内存。
原型：	#include <linux/vmalloc.h> Void *vmalloc (unsigned long size)；
说明：	分配内核内部连续的内存空间。其分配的内存地址在使用方式上与kmalloc( )函数基本相同，但是该函数分配的是物理地址。此外，没有限制可分配的内存大小。调用该函数时，若没有可分配的内存，进程就会进入睡眠状态，因此，该函数不能用在中断服务函数或中断禁止状态。
变量：	• size　将分配的内存大小。
返回值：	成功返回分配内存的起始地址，失败返回NULL。

Vfree ( )函数

功能：	释放已分配的内存。
原型：	#include <linux/vmalloc.h> Void vfree (const void *addr)；
说明：	利用vmalloc( )函数释放已分配的内存。
变量：	• addr　用于释放的内存地址。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

Mempool_create ( )函数

功能：	生成内存池管理结构体的内存。
原型：	#include <linux/mempool.h> mempool_t*mempool_create (int min_nr,mempool_alloc_t *alloc_fn, mempool_free_t*free_fn,void*pool_data)；
说明：	生成内存池管理结构体的内存。
变量：	• min_nr 预分配内存对象的最低数目； • alloc_fn 内存对象分配用户函数的地址； • free_fn 内存对象释放用户函数的地址； • pool_data 向用户函数传送的内存地址。
返回值：	成功返回内存池管理结构体的地址，失败返回NULL。

Mempool_destroy ( )函数

功能：	取消内存池管理对象。
原型：	#include <linux/mempool.h> Void mempool_destroy (mempool_t *pool)；
说明：	取消内存池管理结构体的内存。
变量：	• pool 将取消的内存池结构体的地址。

Mempool_alloc_t ( )函数类型

功能：	分配内存对象的用户函数类型
原型：	#include <linux/mempool.h> Typedef void *(mempool_alloc_t)(int gfp_mask, void *pool_data)；
说明：	在内存池中通过mempool_alloc ( )函数分配内存对象时，调用的用户分配函数。
变量：	• gfp_mask 分配处理标志； • pool_data 管理对象传送的数据地址。
返回值：	成功返回已分配内存对象的起始地址，失败返回NULL。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

mempool_free_t ( )函数类型

功能：	内存池的内存释放用户函数类型
原型：	#include <linux/mempool.h> typedef void (mempool_free_t)(void *element, void *pool_data);
说明：	在内存池中利用mempool_frdd ( )函数释放已分配内存时，调用的用户释放函数。
变量：	• element　将释放的内存对象地址； • pool_data　管理对象传送的数据地址。

Mempool_alloc ( )函数

功能：	在内存池中分配内存对象
原型：	#include <linux/mempool.h> void *mempool_alloc (mempool_t *pool, int gfp_mask);
说明：	从内存池的管理结构体pool中，分配内存对象。分配时，受gfp_mask的影响。
变量：	• pool　内存池管理结构体的地址； • gfp_mask　分配特性标志。
返回值：	成功返回内存对象的地址，失败返回NULL。

mempool_free ( )函数

功能：	向内存池释放已分配的内存对象。
原型：	#include <linux/mempool.h> void mempool_free (void *element, mempool_t *pool);
说明：	在内存池pool中释放已分配内存对象element。
变量：	• element　分配特性标志； • pool　内存池管理结构体的地址。