1. **Kmalloc**

kmalloc分配的内存在物理上是连续的

#include <linux/slab.h>

void \*kmalloc(size\_t size, int flags);

Size：要分配块的大小

Flags：分配标志，标志定义在 <linux/gfp.h>

GFP\_ATOMIC

用来从中断处理和进程上下文之外的其他代码中分配内存. 从不睡眠.

GFP\_KERNEL

内核内存的正常分配. 可能睡眠.

\_\_GFP\_DMA

这个标志要求分配在 DMA 内存区

\_\_GFP\_HIGHMEM

这个标志指示分配的内存位于高端内存区.

...

**Size参数**

内核只能分配固定大小的倍数的内存，如果申请任意大小的内存，得到的肯会多一些，使用kmalloc分配内存存在上下，一般不要超过128kb

1. **高速缓存**

高速缓存是为了反复使用同一大小的内存块而设计的，我们使用高速缓存减少内存申请和释放的开销，高速缓存被称为 slab 分配器

高速缓存使用 kmem\_cache\_t 类型表示

**创建缓存**

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));

Name：缓存名称

Size：每次从缓存分配内存的大小

Offset：页内第一个对象的偏移，一般为0

Flags：控制如何进行分配

SLAB\_NO\_REAP

设置这个标志保护缓存在系统查找内存时被削减. 一般不使用这个标志

SLAB\_HWCACHE\_ALIGN

这个标志需要每个数据对象被对齐到一个缓存行，这个选项可以是一个好的选择

SLAB\_CACHE\_DMA

这个标志要求每个数据对象在 DMA 内存区分配.

constructor 和 destructor 参数：可选函数，前者可以用来初始化新分配的对象，后者可以用来"清理"对象在它们的内存释放回给系统之前.

**从缓存分配内存**

void \*kmem\_cache\_alloc(kmem\_cache\_t \*cache, int flags);

Cache：参数是你之前已经创建的缓存;

Flags：和 kmalloc 的flags相同

**释放分配的内存**

void kmem\_cache\_free(kmem\_cache\_t \*cache, const void \*obj);

**销毁缓存**

int kmem\_cache\_destroy(kmem\_cache\_t \*cache);

**示例：**

/\* 定义一个缓存 \*/

kmem\_cache\_t \*scullc\_cache;

slab 缓存的创建以这样的方式处理( 在模块加载时 ):

/\* 创建缓存 \*/

scullc\_cache = kmem\_cache\_create("scullc", scullc\_quantum,

                                 0, SLAB\_HWCACHE\_ALIGN, NULL, NULL);

if (!scullc\_cache)

{

    scullc\_cleanup();

    return -ENOMEM;

}

这是它如何分配内存量子:

if (!dptr->data[s\_pos])

{

    // 分配一个量子结构大小的内存

    dptr->data[s\_pos] = kmem\_cache\_alloc(scullc\_cache, GFP\_KERNEL);

    if (!dptr->data[s\_pos])

        goto nomem;

    // 内存清零

    memset(dptr->data[s\_pos], 0, scullc\_quantum);

}

还有这些代码行释放内存:

for (i = 0; i < qset; i++)

    if (dptr->data[i])

        // 释放内存

        kmem\_cache\_free(scullc\_cache, dptr->data[i]);

最后, 在模块卸载时, 我们不得不返回缓存给系统:

/\* 销毁缓存 \*/

if (scullc\_cache)

    kmem\_cache\_destroy(scullc\_cache);

1. **Get\_free\_page**

如果需要分配大块内存，应该使用页分配技术

**分配页**

get\_zeroed\_page(unsigned int flags);

返回一个指向新页的指针并且用零填充了该页.

\_\_get\_free\_page(unsigned int flags);

类似于 get\_zeroed\_page, 但是没有清零该页.

\_\_get\_free\_pages(unsigned int flags, unsigned int order);

分配并返回一个指向一个内存区第一个字节的指针, 内存区可能是几个(物理上连续)页长但是没有清零.

flags 参数同 kmalloc 的用法相同;

order 是你在请求的或释放的页数的以 2 为底的对数(即, log2N). 例如, 如果你要1页 则order 为 0, 如果你请求 8 页就是 3.

**释放页**

void free\_page(unsigned long addr);

void free\_pages(unsigned long addr, unsigned long order);

**示例**

下列代码行显示了它如何分配内存

/\* 为量子分配内存 \*/

if (!dptr->data[s\_pos])

{

    // 分配一页

    dptr->data[s\_pos] = (void \*)\_\_get\_free\_pages(GFP\_KERNEL, dptr->order);

    if (!dptr->data[s\_pos])

        goto nomem;

    // 清 0

    memset(dptr->data[s\_pos], 0, PAGE\_SIZE << dptr->order);

}

scullp 中释放内存的代码

/\* 释放量子的内存 \*/

for (i = 0; i < qset; i++)

    if (dptr->data[i])

        // 释放页面

        free\_pages((unsigned long)(dptr->data[i]), dptr->order);

1. **Vmalloc**

和kmalloc类似，但其分配的内存在物理内存上是不连续的

vmalloc 的使用在大部分情况下不鼓励. 从 vmalloc 获得的内存用起来稍微低效些

vmalloc一般用来分配涉及多个页面大小的内存，因为其他分配技术要找多个连续的页面花的时间会比较长

分配与释放函数

#include <linux/vmalloc.h>

void \*vmalloc(unsigned long size);

void vfree(void \* addr);

1. **Pre-cpu变量**

Pre-cpu变量是针对cpu的变量，当你创建一个pre-CPU 变量, 系统中每个处理器获得这个变量拷贝，在运行过程中会禁止内核抢占

示例：

DEFINE\_PER\_CPU(int, name); /\* 为每个CPU定义一个 int 类型的name变量，静态定义 \*/

get\_cpu\_var(name)++; /\* 当前处理器上的name变量 +1 \*/

put\_cpu\_var(name); /\* 完成对name的操作后，激活当前处理器的内核抢占 \*/