内核的内存使用不像用户空间那样随意，内核的内存出现错误时也只有靠自己来解决（用户空间的内存错误可以抛给内核来解决）。

**主要内容：**

* 内存的管理单元
* 获取内存的方法
* 获取高端内存
* 内核内存的分配方式
* 总结

### **1. 内存的管理单元**

内存最基本的管理单元是页，同时按照内存地址的大小，大致分为3个区。

#### **1.1 页**

页的大小与体系结构有关，在 x86 结构中一般是 4KB或者8KB。

可以通过 getconf 命令来查看系统的page的大小：

[wangyubin@localhost ]$ getconf -a | grep -i 'page'

PAGESIZE 4096

PAGE\_SIZE 4096

\_AVPHYS\_PAGES 637406

\_PHYS\_PAGES 2012863

以上的 PAGESIZE 就是当前机器页大小，即 4KB

页的结构体头文件是： <linux/mm\_types.h> 位置：include/linux/mm\_types.h

/\*

 \* 页中包含的成员非常多，还包含了一些联合体

 \*/

struct page

{

    unsigned long flags; /\* 存放页的状态，各种状态参见<linux/page-flags.h> \*/

    atomic\_t \_count;     /\* 页的引用计数 \*/

#if defined(WANT\_PAGE\_VIRTUAL)

    void \*virtual; /\* 页的虚拟地址 \*/

#endif             /\* WANT\_PAGE\_VIRTUAL \*/

....

};

物理内存的每个页都有一个对应的 page 结构

#### **1.2 区**

内核将内存按地址的顺序分成了不同的区，有的硬件只能访问有专门的区。

内核中分的区定义在头文件 <linux/mmzone.h> 位置：include/linux/mmzone.h

内存区的种类参见 enum zone\_type 中的定义。

内存区的结构体定义也在 <linux/mmzone.h> 中。

具体参考其中 struct zone 的定义。

一般主要关注3个区：

|  |  |  |
| --- | --- | --- |
| **区** | **描述** | **物理内存** |
| ZONE\_DMA | DMA使用的页 | <16MB |
| ZONE\_NORMAL | 正常可寻址的页 | 16～896MB |
| ZONE\_HIGHMEM | 动态映射的页 | >896MB |

某些硬件只能直接访问内存地址，不支持内存映射，对于这些硬件内核会分配 ZONE\_DMA 区的内存。

某些硬件的内存寻址范围很广，比虚拟寻址范围还要大的多，那么就会用到 ZONE\_HIGHMEM 区的内存，

对于大部分的内存申请，只要用 ZONE\_NORMAL 区的内存即可。

### **2. 获取内存的方法**

内核中提供了多种获取内存的方法，了解各种方法的特点，可以恰当的将其用于合适的场景。

#### **2.1 按页获取 - 最原始的方法，用于底层获取内存的方式**

以下分配内存的方法参见：<linux/gfp.h>

|  |  |
| --- | --- |
| **方法** | **描述** |
| alloc\_page(gfp\_mask) | 只分配一页，返回指向页结构的指针 |
| alloc\_pages(gfp\_mask, order) | 分配 2^order 个页，返回指向第一页页结构的指针 |
| \_\_get\_free\_page(gfp\_mask) | 只分配一页，返回指向其逻辑地址的指针 |
| \_\_get\_free\_pages(gfp\_mask, order) | 分配 2^order 个页，返回指向第一页逻辑地址的指针 |
| get\_zeroed\_page(gfp\_mask) | 只分配一页，让其内容填充为0，返回指向其逻辑地址的指针 |

alloc\*\* 方法和 get\*\* 方法的区别在于，一个返回的是内存的物理地址，一个返回内存物理地址映射后的逻辑地址。

如果无须直接操作物理页结构体的话，一般使用 get\*\* 方法。

相应的释放内存的函数如下：也是在 <linux/gfp.h> 中定义的

extern void \_\_free\_pages(struct page \*page, unsigned int order);

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

extern void free\_hot\_page(struct page \*page);

在请求内存时，参数中有个 gfp\_mask 标志，这个标志是控制分配内存时必须遵守的一些规则。

gfp\_mask 标志有3类：(所有的 GFP 标志都在 <linux/gfp.h> 中定义)

1. 行为标志 ：控制分配内存时，分配器的一些行为
2. 区标志   ：控制内存分配在那个区(ZONE\_DMA, ZONE\_NORMAL, ZONE\_HIGHMEM 之类)
3. 类型标志 ：由上面2种标志组合而成的一些常用的场景

行为标志主要有以下几种：

|  |  |
| --- | --- |
| **行为标志** | **描述** |
| \_\_GFP\_WAIT | 分配器可以睡眠 |
| \_\_GFP\_HIGH | 分配器可以访问紧急事件缓冲池 |
| \_\_GFP\_IO | 分配器可以启动磁盘I/O |
| \_\_GFP\_FS | 分配器可以启动文件系统I/O |
| \_\_GFP\_COLD | 分配器应该使用高速缓存中快要淘汰出去的页 |
| \_\_GFP\_NOWARN | 分配器将不打印失败警告 |
| \_\_GFP\_REPEAT | 分配器在分配失败时重复进行分配，但是这次分配还存在失败的可能 |
| \_\_GFP\_NOFALL | 分配器将无限的重复进行分配。分配不能失败 |
| \_\_GFP\_NORETRY | 分配器在分配失败时不会重新分配 |
| \_\_GFP\_NO\_GROW | 由slab层内部使用 |
| \_\_GFP\_COMP | 添加混合页元数据，在 hugetlb 的代码内部使用 |

区标志主要以下3种：

|  |  |
| --- | --- |
| **区标志** | **描述** |
| \_\_GFP\_DMA | 从 ZONE\_DMA 分配 |
| \_\_GFP\_DMA32 | 只在 ZONE\_DMA32 分配 (**注1**) |
| \_\_GFP\_HIGHMEM | 从 ZONE\_HIGHMEM 或者 ZONE\_NORMAL 分配 (**注2**) |

**注1：**ZONE\_DMA32 和 ZONE\_DMA 类似，该区包含的页也可以进行DMA操作。   
         唯一不同的地方在于，ZONE\_DMA32 区的页只能被32位设备访问。   
**注2：**优先从 ZONE\_HIGHMEM 分配，如果 ZONE\_HIGHMEM 没有多余的页则从 ZONE\_NORMAL 分配。

类型标志是编程中最常用的，在使用标志时，应首先看看类型标志中是否有合适的，如果没有，再去自己组合 行为标志和区标志。

|  |  |  |
| --- | --- | --- |
| **类型标志** | **实际标志** | **描述** |
| GFP\_ATOMIC | \_\_GFP\_HIGH | 这个标志用在中断处理程序，下半部，持有自旋锁以及其他不能睡眠的地方 |
| GFP\_NOWAIT | 0 | 与 GFP\_ATOMIC 类似，不同之处在于，调用不会退给紧急内存池。  这就增加了内存分配失败的可能性 |
| GFP\_NOIO | \_\_GFP\_WAIT | 这种分配可以阻塞，但不会启动磁盘I/O。  这个标志在不能引发更多磁盘I/O时能阻塞I/O代码，可能会导致递归 |
| GFP\_NOFS | (\_\_GFP\_WAIT ｜ \_\_GFP\_IO) | 这种分配在必要时可能阻塞，也可能启动磁盘I/O，但不会启动文件系统操作。  这个标志在你不能再启动另一个文件系统的操作时，用在文件系统部分的代码中 |
| GFP\_KERNEL | (\_\_GFP\_WAIT ｜ \_\_GFP\_IO ｜ \_\_GFP\_FS ) | 这是常规的分配方式，可能会阻塞。这个标志在睡眠安全时用在进程上下文代码中。  为了获得调用者所需的内存，内核会尽力而为。这个标志应当为首选标志 |
| GFP\_USER | (\_\_GFP\_WAIT ｜ \_\_GFP\_IO ｜ \_\_GFP\_FS ) | 这是常规的分配方式，可能会阻塞。用于为用户空间进程分配内存时 |
| GFP\_HIGHUSER | (\_\_GFP\_WAIT ｜ \_\_GFP\_IO ｜ \_\_GFP\_FS )｜\_\_GFP\_HIGHMEM) | 从 ZONE\_HIGHMEM 进行分配，可能会阻塞。用于为用户空间进程分配内存 |
| GFP\_DMA | \_\_GFP\_DMA | 从 ZONE\_DMA 进行分配。需要获取能供DMA使用的内存的设备驱动程序使用这个标志  通常与以上的某个标志组合在一起使用。 |

以上各种类型标志的使用场景总结：

|  |  |
| --- | --- |
| **场景** | **相应标志** |
| 进程上下文，可以睡眠 | 使用 GFP\_KERNEL |
| 进程上下文，不可以睡眠 | 使用 GFP\_ATOMIC，在睡眠之前或之后以 GFP\_KERNEL 执行内存分配 |
| 中断处理程序 | 使用 GFP\_ATOMIC |
| 软中断 | 使用 GFP\_ATOMIC |
| tasklet | 使用 GFP\_ATOMIC |
| 需要用于DMA的内存，可以睡眠 | 使用 (GFP\_DMA｜GFP\_KERNEL) |
| 需要用于DMA的内存，不可以睡眠 | 使用 (GFP\_DMA｜GFP\_ATOMIC)，或者在睡眠之前执行内存分配 |

#### **2.2 按字节获取 - 用的最多的获取方法**

这种内存分配方法是平时使用比较多的，主要有2种分配方法：kmalloc()和vmalloc()

kmalloc的定义在 <linux/slab\_def.h> 中

/\*\*

\* @size - 申请分配的字节数

\* @flags - 上面讨论的各种 gfp\_mask

\*/

static \_\_always\_inline void \*kmalloc(size\_t size, gfp\_t flags)

vmalloc的定义在 mm/vmalloc.c 中

/\*\*

\* @size - 申请分配的字节数

\*/

void \*vmalloc(unsigned long size)

kmalloc 和 vmalloc 区别在于：

kmalloc 分配的内存物理地址是连续的，虚拟地址也是连续的

vmalloc 分配的内存物理地址是不连续的，虚拟地址是连续的

因此在使用中，用的较多的还是 kmalloc，因为kmalloc 的性能较好。

kmalloc 和 vmalloc 所对应的释放内存的方法分别为：

void kfree(const void \*)

void vfree(const void \*)

#### **2.3 slab层获取 - 效率最高的获取方法**

频繁的分配/释放内存必然导致系统性能的下降，所以有必要为频繁分配/释放的对象内心建立缓存。

而且，如果能为每个处理器建立专用的高速缓存，还可以避免 SMP锁带来的性能损耗。

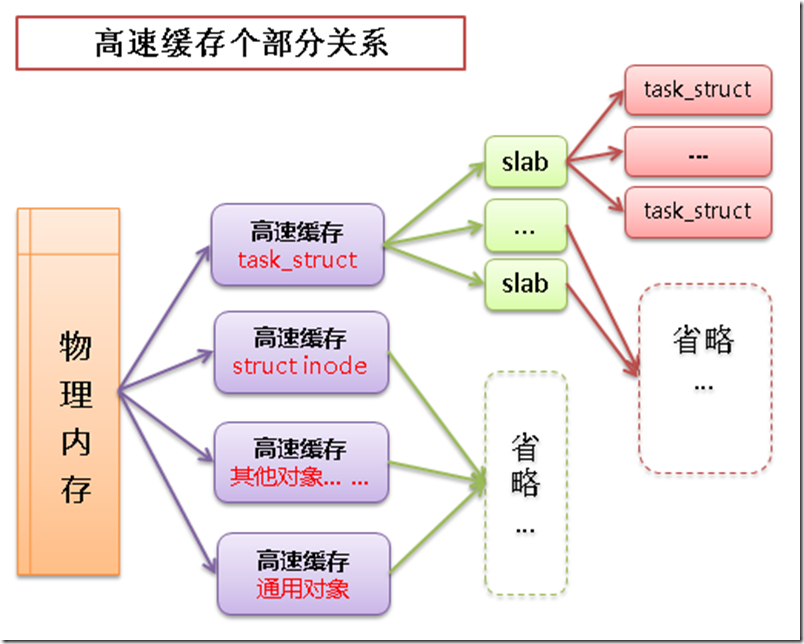
##### **2.3.1 slab层实现原理**

linux中的高速缓存是用所谓 slab 层来实现的，slab层即内核中管理高速缓存的机制。

整个slab层的原理如下：

1. 可以在内存中建立各种对象的高速缓存(比如进程描述相关的结构 task\_struct 的高速缓存)
2. 除了针对特定对象的高速缓存以外，也有通用对象的高速缓存
3. 每个高速缓存中包含多个 slab，slab用于管理缓存的对象
4. slab中包含多个缓存的对象，物理上由一页或多个连续的页组成

高速缓存->slab->缓存对象之间的关系如下图：

[](https://images0.cnblogs.com/blog/83005/201305/23223357-705e9a54bae54639906aa76880012aa5.png)

##### **2.3.2 slab层的应用**

slab结构体的定义参见：mm/slab.c

struct slab {

struct list\_head list; /\* 存放缓存对象，这个链表有 满，部分满，空 3种状态 \*/

unsigned long colouroff; /\* slab 着色的偏移量 \*/

void \*s\_mem; /\* 在 slab 中的第一个对象 \*/

unsigned int inuse; /\* slab 中已分配的对象数 \*/

kmem\_bufctl\_t free; /\* 第一个空闲对象(如果有的话) \*/

unsigned short nodeid; /\* 应该是在 NUMA 环境下使用 \*/

};

slab层的应用主要有四个方法：

/\*\*

\* 创建高速缓存

\* 参见文件： mm/slab.c

\* 这个函数的注释很详细，这里就不多说了。

\*/

struct kmem\_cache \*kmem\_cache\_create (const char \*name, size\_t size, size\_t align,

unsigned long flags, void (\*ctor)(void \*))

/\*\*

\* 从高速缓存中分配对象也很简单

\* 函数参见文件：mm/slab.c

\* @cachep - 指向高速缓存指针

\* @flags - 之前讨论的 gfp\_mask 标志，只有在高速缓存中所有slab都没有空闲对象时，

\* 需要申请新的空间时，这个标志才会起作用。

\*

\* 分配成功时，返回指向对象的指针

\*/

void \*kmem\_cache\_alloc(struct kmem\_cache \*cachep, gfp\_t flags)

/\*\*

\* 向高速缓存释放对象

\* @cachep - 指向高速缓存指针

\* @objp - 要释放的对象的指针

\*/

void kmem\_cache\_free(struct kmem\_cache \*cachep, void \*objp)

/\*\*

\* 销毁高速缓存

\* @cachep - 指向高速缓存指针

\*/

void kmem\_cache\_destroy(struct kmem\_cache \*cachep)

我做了创建高速缓存的例子，来尝试使用上面的几个函数。

#include <linux/init.h>

#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/time.h>

#include <linux/string.h>

#include <linux/slab.h>

#include <linux/slab\_def.h>

MODULE\_LICENSE("Dual BSD/GPL");

#define MYSLAB "testslab"

static void print\_student(struct student \*);

// 高速缓存

static struct kmem\_cache \*myslab;

// 测试对象

struct student

{

    int id;

    char \*name;

};

/\* 申请内存时调用的构造函数 \*/

static void ctor(void \*obj)

{

    printk(KERN\_ALERT "constructor is running....\n");

}

static int testslab\_init(void)

{

struct student \*stu1, \*stu2;

    /\* 建立slab高速缓存，名称就是宏 MYSLAB \*/

myslab = kmem\_cache\_create(MYSLAB, sizeof(struct student), 0, 0, ctor);

    /\* 高速缓存中分配2个对象 \*/

    printk(KERN\_ALERT "alloc one student....\n");

    // 高速缓存中分配一个对象空间

    stu1 = (struct student \*)kmem\_cache\_alloc(myslab, GFP\_KERNEL);

    stu1->id = 1;

    stu1->name = "wyb1";

print\_student(stu1);

    printk(KERN\_ALERT "alloc one student....\n");

    stu2 = (struct student \*)kmem\_cache\_alloc(myslab, GFP\_KERNEL);

    stu2->id = 2;

    stu2->name = "wyb2";

print\_student(stu2);

    /\* 释放高速缓存中的对象 \*/

    printk(KERN\_ALERT "free one student....\n");

kmem\_cache\_free(myslab, stu1);

    printk(KERN\_ALERT "free one student....\n");

kmem\_cache\_free(myslab, stu2);

    /\* 执行完后查看 /proc/slabinfo 文件中是否有名称为 “testslab”的缓存 \*/

    return 0;

}

static void testslab\_exit(void)

{

    /\* 删除建立的高速缓存 \*/

    kmem\_cache\_destroy(myslab);

    printk(KERN\_ALERT "testslab is exited!\n");

printk(KERN\_ALERT "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

    /\* 执行完后查看 /proc/slabinfo 文件中是否有名称为 “testslab”的缓存 \*/

}

static void print\_student(struct student \*stu)

{

    if (stu != NULL)

    {

        printk(KERN\_ALERT "\*\*\*\*\*\*\*\*\*\*student info\*\*\*\*\*\*\*\*\*\*\*\n");

        printk(KERN\_ALERT "student id   is: %d\n", stu->id);

        printk(KERN\_ALERT "student name is: %s\n", stu->name);

        printk(KERN\_ALERT "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

    }

    else

        printk(KERN\_ALERT "the student info is null!!\n");

}

module\_init(testslab\_init);

module\_exit(testslab\_exit);

### **3. 获取高端内存**

高端内存就是之前提到的 ZONE\_HIGHMEM 区的内存。

在x86体系结构中，这个区的内存不能映射到内核地址空间上，也就是没有逻辑地址，

为了使用 ZONE\_HIGHMEM 区的内存，内核提供了永久映射和临时映射2种手段：

#### 

#### **3.1 永久映射**

永久映射的函数是可以睡眠的，所以只能用在进程上下文中。

/\* 将 ZONE\_HIGHMEM 区的一个page永久的映射到内核地址空间

\* 返回值即为这个page对应的逻辑地址

\*/

static inline void \*kmap(struct page \*page)

/\* 允许永久映射的数量是有限的，所以不需要高端内存时，应该及时的解除映射 \*/

static inline void kunmap(struct page \*page)

#### **3.2 临时映射**

临时映射不会阻塞，也禁止了内核抢占，所以可以用在中断上下文和其他不能重新调度的地方。

/\*\*

\* 将 ZONE\_HIGHMEM 区的一个page临时映射到内核地址空间

\* 其中的 km\_type 表示映射的目的，

\* enum kn\_type 的定义参见：<asm/kmap\_types.h>

\*/

static inline void \*kmap\_atomic(struct page \*page, enum km\_type idx)

/\* 相应的解除映射是个宏 \*/

#define kunmap\_atomic(addr, idx) do { pagefault\_enable(); } while (0)

以上的函数都在 <linux/highmem.h> 中定义的。

### **4. 内核内存的分配方式**

#### **4.1 进程内核栈上的静态分配**

在x86体系结构中，内核栈的大小一般就是1页或2页，即 4KB ~ 8KB

内核栈可以在编译内核时通过配置选项将内核栈配置为1页，

当有中断发生时，如果共享内核栈，中断程序和被中断程序共享一个内核栈会可能导致空间不足，

于是，每个进程除了有个内核栈之外，还有一个中断栈，中断栈一般也就1页大小。

查看当前系统内核栈大小的方法：

[xxxxx@localhost ~]$ ulimit -a | grep 'stack'

stack size (kbytes, -s) 8192

#### **4.2 按CPU分配**

与单CPU环境不同，SMP（多cpu）环境下的并行是真正的并行。单CPU环境是宏观并行，微观串行。

真正并行时，会有更多的并发问题。

假定在多线程处理中，有如下场景：

void\* p;

if (p == NULL)

{

/\* 对 P 进行相应的操作，最终 P 不是NULL了 \*/

}

else

{

/\* P 不是NULL，继续对 P 进行相应的操作 \*/

}

在单CPU环境下，上述情况无需加锁，只需在 if 处理之前禁止内核抢占，在 else 处理之后恢复内核抢占即可。

而在SMP环境下，上述情况必须加锁，因为禁止内核抢占只能禁止当前CPU的抢占，其他的CPU仍然调度线程B 来抢占线程A 的执行

SMP环境下加锁过多的话，会严重影响并行的效率，如果是自旋锁的话，还会浪费其他CPU的执行时间。

所以内核中才有了按CPU分配数据的接口。

按CPU分配数据之后，每个CPU自己的数据不会被其他CPU访问，虽然浪费了一点内存，但是会使系统更加的简洁高效。

##### **4.2.2 编译时分配**

可以在编译时就定义分配给每个CPU的变量，其分配的接口参见：<linux/percpu-defs.h>

/\* 给每个CPU声明一个类型为 type，名称为 name 的变量 \*/

DECLARE\_PER\_CPU(type, name)

/\* 给每个CPU定义一个类型为 type，名称为 name 的变量 \*/

DEFINE\_PER\_CPU(type, name)

注意上面两个宏，一个是声明，一个是定义。

分配好变量后，就可以在代码中使用这个变量 name 了。

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

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 \*/

##### **4.2.3 运行时分配**

除了像上面那样静态的给每个CPU分配数据，还可以以指针的方式在运行时给每个CPU分配数据。

动态分配参见：<linux/percpu.h>

/\* 给每个处理器分配一个 size 字节大小的对象，对象的偏移量是 align \*/

extern void \*\_\_alloc\_percpu(size\_t size, size\_t align);

/\* 释放所有处理器上已分配的变量 \_\_pdata \*/

extern void free\_percpu(void \*\_\_pdata);

/\* 还有一个宏，是按对象类型 type 来给每个CPU分配数据的，

\* 其实本质上还是调用了 \_\_alloc\_percpu 函数 \*/

#define alloc\_percpu(type) (type \*)\_\_alloc\_percpu(sizeof(type), \

\_\_alignof\_\_(type))

动态分配的一个使用例子如下：

void \*percpu\_ptr;

unsigned long \*foo;

percpu\_ptr = alloc\_percpu(unsigned long);

if (!percpu\_ptr)

/\* 内存分配错误 \*/

foo = get\_cpu\_var(percpu\_ptr);/\* 操作foo ... \*/

put\_cpu\_var(percpu\_ptr);

### **5. 总结**

在众多的内存分配函数中，如何选择合适的内存分配函数很重要，下面总结了一些选择的原则：

|  |  |
| --- | --- |
| **应用场景** | **分配函数选择** |
| 如果需要物理上连续的页 | 选择低级页分配器或者 kmalloc 函数 |
| 如果kmalloc分配是可以睡眠 | 指定 GFP\_KERNEL 标志 |
| 如果kmalloc分配是不能睡眠 | 指定 GFP\_ATOMIC 标志 |
| 如果不需要物理上连续的页 | vmalloc 函数 (vmalloc 的性能不如 kmalloc) |
| 如果需要高端内存 | alloc\_pages 函数获取 page 的地址，在用 kmap 之类的函数进行映射 |
| 如果频繁撤销/创建教导的数据结构 | 建立slab高速缓存 |