第32卷 第24期 计 算 机 工 程 2006年12月
Vol.32 No.24 Computer Engineering December 2006
·软件技术与数据库· 文章编号：1000—3428(2006)24—0031—03 文献标识码：A 中图分类号：TP316

# Linux 内核 Slab 内存缓冲区管理器

黄应江，陈移风
(华南理工大学计算机科学与工程学院，广州 510640)

**摘 要：**Linux 是开源操作系统，内核运行时，会使用很多数据结构对象，即内核对象。如何高效地管理这些对象的创建、使用和回收，需一整套完善的管理机制。该文介绍了 Slab 内存缓冲区管理器的原理，以及 Linux 怎样使用 Slab 管理器管理内核中各种各样的内核对象。
**关键词：**Slab 管理器；对象；缓冲区

# Linux Kernel Slab Memory Buffer Manager

HUANG Yingjiang, CHEN Yifeng
(School of Computer Science & Engineering, South China University of Technology, Guanzhu 510640)

**【Abstract】** The source codes of Linux OS are open. When Linux kernel is running, it uses a lot of structural objects , called kernel objects. How to manage the creation, usage and recycle of these objects requires a perfect management mechanism. This paper discusses the theory of Slab memory buffer manager and how to use Slab manager to manage all kinds of kernel objects in Linux kernel.
**【Key words】** Slab manager; Object; Buffer

随着 Linux 操作系统越来越普及和广泛应用，尤其在嵌入式领域越来越受关注，越来越多的人开始研究 Linux 内核。Linux 是开放源代码的，所以在中国更加应该受到关注，对于促进中国开发自主的操作系统具有积极的意义。现在，希望越来越多的人投入到这个热潮中来。本人接触这方面的研究项目，成为这个热潮中积极的一员，在研究学习过程中，遇到很多困难。很多人也应该会遇到这样那样的问题，所以把学习中总结归纳出的一些东西与大家共同探讨。下面介绍 Linux 分配管理内核中的数据结构(或对象)的方法——Slab 缓冲区管理器。

## 1 为什么选择 Slab

Linux 内核在运行中常常需要动态地分配一些缓冲区。比如当打开一个文件时需要建立与文件磁盘索引节点对应的 struct inode 类型的内存索引节点数据结构，还有与磁盘目录项对应的 struct dentry 类型的内存目录项数据结构。这些数据结构在运行中根据需要动态建立，不像内核代码和内核全局变量既不需要分配，也不需要释放，是静态的。这些小块存储空间是动态变化的，运行中就会有申请内存与释放内存的问题。

那么小对象存储空间的管理，该用什么方法呢？先考虑下面几种情况。

对于这些小块存储空间，如果按照常规的内存分配方法调用 alloc_pages ()整页整页的分配，由于根本用不了这么多空间，这样就造成浪费。对于这种情况，何不在分配的内存页中放几个小对象呢？

每次分配的小对象存储空间一般都要进行初始化(如队列头初始化，因为这些小对象一般都会加入若干队列)。如果这些对象释放后，不释放占用的页面内存，下次分配同类对象时就可以重用这些对象，从而不需初始化，那就可以提高内核效率。

针对小对象存储空间的使用问题，Linux2.4.0 采用了一种称为“ Slab ”的缓冲区分配和管理方法。

## 2 Slab 管理器原理

在 slab 方法中，每种重要的数据结构都有自己的专用缓冲区队列，每种数据结构都有相应的“ 构造 ”( constructor )和“ 析构 ”( destructor )函数。同时，还借用面向对象程序设计技术中的名词，不再称“ 结构 ”而称“ 对象 ”( object )。缓冲区队列中的各个对象在建立时用其“ 构造 ”函数进行初始化，所以一经分配立即就能使用，而在释放时则恢复成原状。每个队列中对象的个数是动态变化的，不够时可以增添。同时，又定期地检查，将有富余的队列加以精简。

此外，Slab 管理方法还有个特点，每种对象的缓冲区队列并非由各个对象直接构成，而是由一连串的“ 大块 ”( slab )构成，而每个 slab 块中则包含了若干同种对象。每个这样的一个 slab 块由一个 slab_t 类型的数据结构描述，而且，这个 slab_t 数据结构本身也放在 slab 块中(后面将会提到，对于大对象，为了充分利用 slab 空间，把 slab 描述结构和对象链接数组存放在通用缓冲区中)。图 1 就是 slab 块结构示意图。

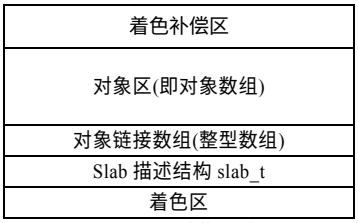


**图 1 slab 结构示意图**

每个 slab 可能由 2 的幂次方个(最多 32 个)连续的物理页面构成，具体大小因对象大小而异。每个 slab 上都有一个 slab

**作者简介：**黄应江(1975－)，男，硕士，主研方向：嵌入式 Linux 操作系统移植与内核技术，嵌入式 Linux 非 PCI 的 USB 总线驱动和非 PCI 的网卡驱动技术；陈移风，副教授
**收稿日期：**2005-12-25 **E-mail：**huangyj@scut.edu.cn

描述结构，描述它所在的 slab，提供 slab 的管理信息。用于同一种对象的多个 slab 通过 slab 描述结构中的 list 字段链接在一起。而这些 slab 分成 3 截：第 1 截是各个 slab 上所有对象都已分配使用；第 2 截是各个 slab 上的对象部分地分配使用；第 3 截是各个 slab 上的对象全部是空闲对象。每个 slab 上有一个对象区，这就是对象存放的地方，是对象数据结构的数组。每个 slab 上还有一个对象链接数组，实际上是一个整型数组，用来实现一个空闲对象链，与对象区的对象数组对应。在 slab 描述结构中有一个 free 整型字段，表明第 1 个空闲对象下标。free 字段和对象链接数组就构成了一个空闲对象链。比如 free 为 5，则第 1 个空闲对象下标为 5，如果对象链接数组下标为 5 的元素的值为 3，则下一个空闲对象就是对象区下标为 3 的对象。最后一个空闲对象下标对应的对象链接数组中的元素的值为 BUFCTL(0xffffffff)。slab 最前面是着色区，这是为了使 slab 中的每个对象起始地址按高速缓存中的缓冲行大小对齐而留下的空白区。slab 最后面是着色补偿区。

## 3 Salb 相关数据结构

下面是 slab 描述结构 slab_t 的定义：

```
typedef struct slab_s
{
  struct list_head      list;
  unsigned long colouroff;
  void                  *s_mem;       /* including colour offset */
  unsigned int          inuse;        /* num of objs active in slab */
  kmem_bufctl_t         free;
} slab_t;
```

字段说明：

list：每种对象的缓冲区都由若干 slab 块构成，这些 slab 块通过 list 字段构成 slab 链；

colouroff：着色区大小，实际上包括着色区大小、slab_t 描述结构大小和对象链接数组大小，即为第 1 个对象距离 slab 开始地址的距离；

s_mem：指向第 1 个对象；

inuse：本 slab 中已经分配使用的对象数；

free：第 1 个空闲对象在对象区的下标。

为每种对象建立的 slab 队列都有个队列头，其控制结构数据类型为 kmem_cache_t 。该数据结构中除用来维持 slab 队列的各种指针外，还记录了适用于队列中每个 slab 的各种参数，以及两个函数指针：一个是对象的构造函数，另一个是析构函数。

队列头控制结构对象的数据类型为 kmem_cache_t(这个数据结构字段较多，由于篇幅有限，这里就不一一列出，请参考源代码文件：mm\slab.c)，如下：

```
struct kmem_cache_s {
  … //字段定义请参考源代码文件：mm\slab.c
  };
typedef struct kmem_cache_s     kmem_cache_t;
```

主要字段说明：

slabs：slab 队列的队列头，每种对象的缓冲区由若干个 slab 构成，这些 slab 链成一个队列；

firstnotfull：指向第一个含有空闲对象的 slab；

objsize：对象大小；

num：每个 slab 中的对象数；

gfporder：每个 slab 的页面数为 2 的 gfporder 次方；

slabp_cache：当 slab_t 描述数据结构和对象链接数组不在 slab 中时，它指向存放 slab_t 描述结构和对象链接数组的通用缓冲区队列的队列头；

ctor，dtor：对象的构造和析构函数指针；

colour，colour_off ，colour_next：着色区大小计算的几个字段。

每个这样的队头需要一个 kmem_cache_t 数据结构，可见这个队头也是动态的，对这些队头的管理内核也是通过 slab 队列来管理的，其队头又是一个 kmem_cache_t 数据结构，这个数据结构的内核变量名为 cache_cache，是内核中的一个全局变量。

这样，就形成了一种层次式的树形结构：

总根 cache_cache 是一个 kmem_cache_t 结构，用来维持第 1 层 slab 队列，这些 slab 上对象都是 kmem_cache_t 数据结构。第 1 层 slab 上的每个对象，即 kmem_cache_t 数据结构都是队头，用来维持一个第 2 层 slab 队列。第 2 层 slab 队列基本上都是某种对象，即数据结构专用的队列。内核中 cache_cache 定义如下：

```
static kmem_cache_t cache_cache = {
  slabs:        LIST_HEAD_INIT(cache_cache.slabs),
  firstnotfull: &cache_cache.slabs,
  objsize:      sizeof(kmem_cache_t),
  flags:        SLAB_NO_REAP,
  spinlock:     SPIN_LOCK_UNLOCKED,
  colour_off:   L1_CACHE_BYTES,
  name:         "kmem_cache",
};
```

总体的组织如图 2 所示。

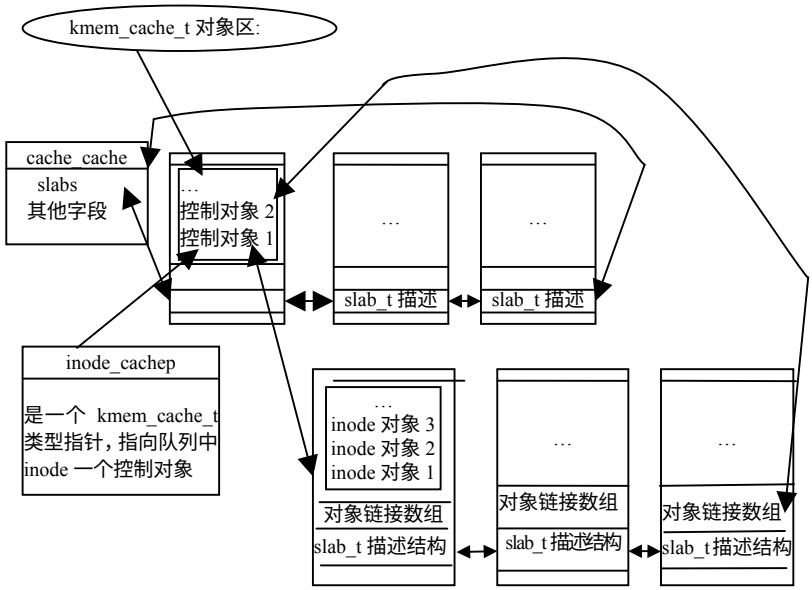


**图 2 对象缓冲区结构示意图**

第 1 层是一个 kmem_cache_t 控制对象的 slab 队列，里面存放的对象是一些 kmem_cache_t 对象，队列头是 cache_cache 全局变量，这个变量本身也是一个 kmem_cache_t 类型的控制对象。第 1 层队列中存放的每一个对象

kmem_cache_t 是一个第 2 层 slab 队列的队列头，比如图 2 所示 kmem_cache_t 类型指针 inode_cachep 所指向的就是一个第 1 层队列中的一个控制对象，而这个对象又是内存索引节点 slab 队列的队列头。图中举了一个内核比较常用的对象的缓冲区队列 inode_cachep，这就是内存索引节点的缓冲区队列头指针，指向在第 1 层 slab 队列中分配的索引节点控制结构队列头。

slab 管理方法中对象的分配和回收不是以单个对象进行的，而是以 slab 为单位进行的。

当内核中分配某个对象的空间时，首先在 slab 队列中查找空闲对象，如果没有空闲对象了，就分配队列头控制结构中字段 gfporder 指定的 2 的 gfporder 次方个物理页面，这些页面构成一个 slab，初始化这个 slab，包括调用队列头中指定的构造函数初始化对象区中的各个对象，并把这个 slab 链入 slab 队列，以后就可以在这个 slab 中分配空闲对象了。

回收时，内核在 slab 队列中从后往前(因为空闲 slab 在队尾)查看每个队列，如果 slab 描述结构中字段 inuse 为 0，就表明这个 slab 是空闲的，可以回收占用的物理内存页面，同时还要对 slab 中的每个对象调用析构函数，并从 slab 队列中脱链。

## 4 通用缓冲区

内核中对一些常用的数量比较大的数据结构建立 slab 专用缓冲区队列，但是对一些很少使用而且数量比较少的数据结构就不适宜建立专门的缓冲区队列。为了管理这种数据结构，内核中建立了 13*2 个通用缓冲区队列，包含几何分布的缓冲存储区，这些缓冲区的对象分别是 32B、64B、128B，…，128KB，每种大小的缓冲区又有两个队列：一个用于普通对象，一个用于与 DMA 传送相关的对象。当然，通用缓冲区的队列头控制结构对象也是第 1 层 slab 队列中的对象，存放在第一层 slab 队列中。内核中有一个全局数组 cache_sizes，数据类型为：

```
typedef struct cache_sizes {
    size_t          cs_size;
    kmem_cache_t    *cs_cachep;
    kmem_cache_t    *cs_dmacachep;
} cache_sizes_t;
static cache_sizes_t cache_sizes[] = {…};
```

一个元素代表一种长度的通用缓冲区，cs_size 为队列中对象大小，cs_cachep 指向用于分配普通对象的控制队列头，cs_dmacachep 指向用于分配与 DMA 传送相关的对象的队列头。这些控制结构队列头的分配是内核在初始化时就在 cache_cache 第 1 层队列中分配好了的。

对于大对象(如大于页面大小的 1/8)，把 slab_t 描述结构和对象链接数组跟对象一起存放在 slab 中就不适宜了，这时就把对象链接数组和 slab_t 描述结构作为一个整体，看作另一种对象，内核在适当大小的通用缓冲区中分配存放这种对象的空间。在这个大对象的队列头的控制对象中有一个 kmem_cache_t 类型数据结构指针字段 slabp_cache 指向适当大小的通用缓冲区的队列头控制对象。当分配一个大对象的 slab 时，就在 slabp_cache 指向的通用缓冲区中分配一个对象空间存放 slab_t 描述结构和对象链接数组。同样，同一个对象的各个 slab 的 slab_t 描述结构通过 list 字段链接起来。

## 5 结束语

本文结合 Linux，介绍了 Slab 缓冲区管理器的原理以及在 Linux 内核中的广泛应用。内核中需要动态分配许多常用或不常用的数据结构，这些数据结构经过分配、使用和回收的过程，需要一种高效管理方法，Slab 管理器正是这样一种方法。

### 参考文献


1 胡希明, 毛德操. Linux 内核源代码情景分析[M]. 杭州: 浙江大学出版社, 2001.

2 Bovet D P, Cesati M. 深入理解 Linux 内核(第 2 版)[M]. 北京: 中国电力出版社, 2004.


(上接第 16 页)

## 3 安全性分析

在本文的模幂运算实现方法中，模幂运算被分解成随机的原子操作序列，这种随机性主要来源于 2 个方面：(1)每执行一次完整的模乘和模平方操作时，本文会插入随机数目的无效原子操作，并且这些原子操作的操作数和运算结果目标地址也是完全随机的；(2)在任务调度过程中，从 MM、SQR、IVD 的原子操作序列中随机选取后续操作，MM、SQR、IVD 操作随机交替执行，各操作的具体执行时间也是完全随机的。

模乘和模平方操作的目标地址完全随机；任务调度和原子操作的执行是并行的；并且操作序列与系数 $d$ 之间不存在直接相关性。总之，本文的实现方法能够有效地防止 SPA 和 DPA 攻击。

## 4 性能分析

本文以完成一次模幂运算所需运算时间来衡量模幂实现方法的运算性能。在实现方法中，模幂运算被分解成随机的原子操作序列，并且是各次随机的。为此，本文以平均运算时间来衡量运算性能。

假设系数$d_i(0 \le i \le l\text{-}1)$以等概率取 0 或 1，完成一次模幂运算所需的运算时间$t$如下式所示：

$$t = l\times\mathrm{SQR} + (l/2)\times\mathrm{MM} + (l/2)\times\mathrm{IVD}$$

根据第 2 节中关于 IVD 的定义，上式可以转化为

$$t = \left[\frac{3l}{2}m + \frac{l}{2}(2^{f-1}-1)\right]\mathrm{Atom}$$

## 5 结束语

本文提出了一种简单有效的防功耗分析的模幂实现方法，该方法将模乘操作分解成更小粒度的原子操作也就是最小可调度单位，给出了一种将模幂运算分解成随机原子操作序列的任务调度方法。

### 参考文献


1 Kocher P, Jaffe J, Jun B. Differential Power Analysis[C]. Proc. of Advances in Cryptology-CRYPTO'99. Berlin: Springer-Verlag, 1999: 372-387.

2 Messerges T S, Dabbish E A, Sloan R H. Power Analysis Attacks of Modular Exponentiation in Smartcards[C]. Proc. of Cryptographic Hardware and Embedded Systems. Springer, 2000: 144-157.

3 Kocher P C, Jaffe J M. Secure Modular Exponentiation with Leak Minimization for Smartcards and Other Cryptosystems[P]. United States Patent: 6298442B1. 2001-10.

4 A Tenca L F, Todorov G, Koç Ç K. High-radix Design of a Scalable Modular Multiplier[C]. Proc. of Cryptographic Hardware and Embedded Systems. Berlin: Springer Verlag, 2001: 189-205.