[input子系统]

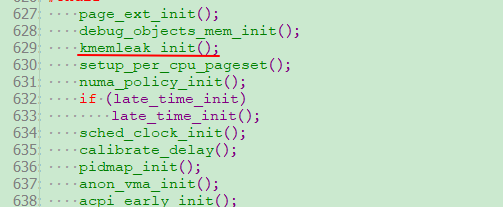
函数列表

|  |
| --- |
| * void input\_set\_capability(struct input\_dev \*dev, unsigned int type, unsigned int code); * int \_\_must\_check input\_register\_device(struct input\_dev \*); * static inline void input\_report\_key(struct input\_dev \*dev, unsigned int code, int value) * static inline void input\_report\_rel(struct input\_dev \*dev, unsigned int code, int value) * static inline void input\_sync(struct input\_dev \*dev) |

|  |
| --- |
| struct input\_polled\_dev {  } |

文件系统：[\fs\seq\_file.c] 作用

asmlinkage \_\_visible void \_\_init start\_kernel(void)



void **\_\_init kmemleak\_init(void)** [ \mm\kmemleak.c]

[\mm\bootmem.c]

[slab 设计思路]

[slab 机制框图] slab 描述符，slab 节点，

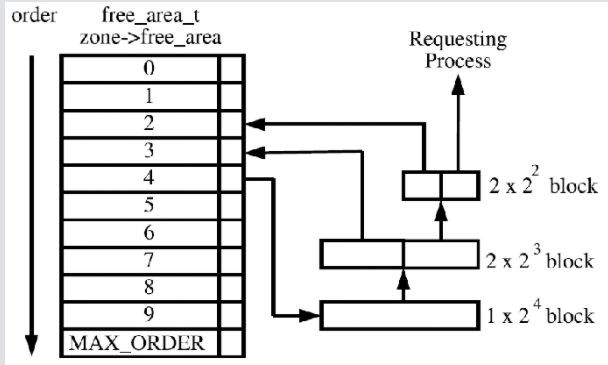
slabs\_partial 链表

slabs\_full 链表

slabs\_free 链表

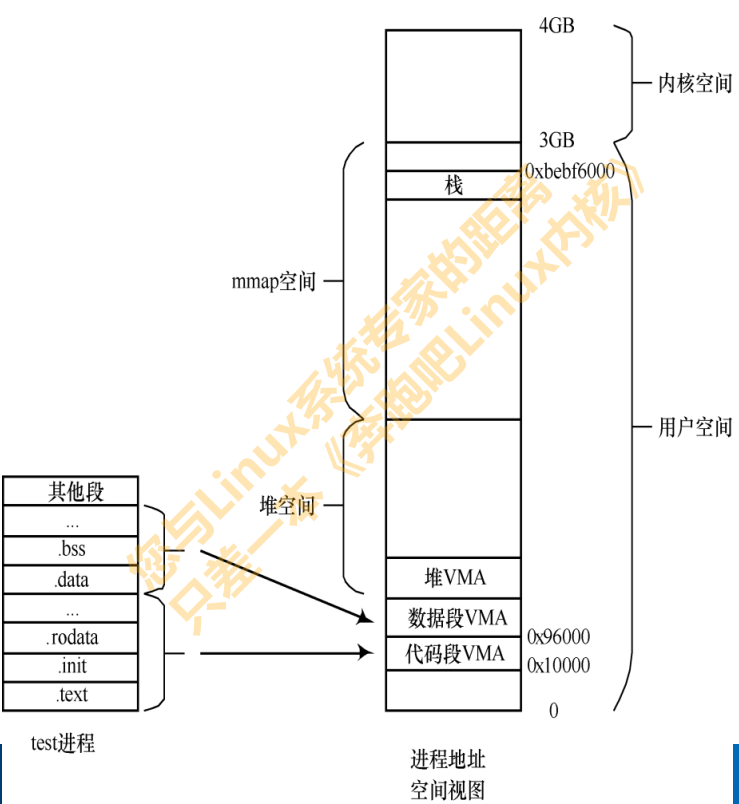
新版本伙伴

Linux内核通过伙伴算法来管理物理内存。伙伴系统（Buddy System）在理论上是非常简单的内存分配算法。它的用途主要是尽可能减少外部碎片，同时允许快速分配与回收物理页面。为了减少外部碎片，连续的空闲页面，根据空闲块（由连续的空闲页面组成）大小，组织成不同的链表（或者orders）。这样所有的2个页面大小的空闲块在一个链表中，4个页面大小的空闲块在另外一个链表中，以此类推。



回收机制

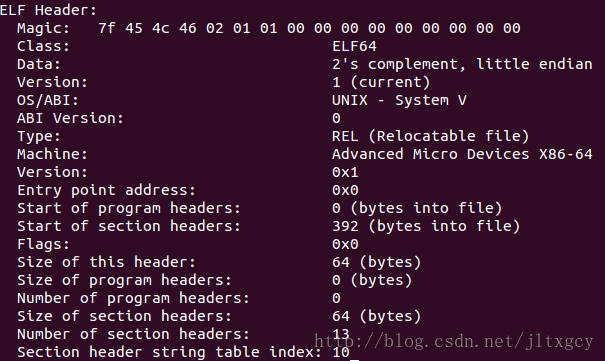
**从进程解决看内存管理**

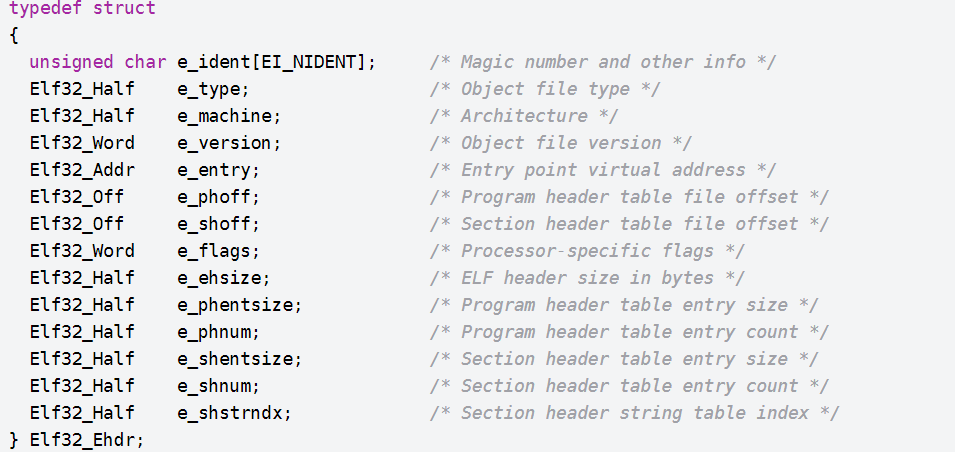


下面我们来利用命令来分析ELF文件结构的每个部分：

1、ELF Header

使用命令readelf -h SimpleSection.o，得到下图。





Type：ELF文件类型，本例中为REL（Relocatable File），可重定位文件。

Start of section headers，段表在文件中偏移，就是图2中Section Table的位置为392（0x188）。

    Size of section headers，ELF文件头的大小为64个字节。

    Number of section headers，ELF拥有多少个断，本例为13个段。见图 7。

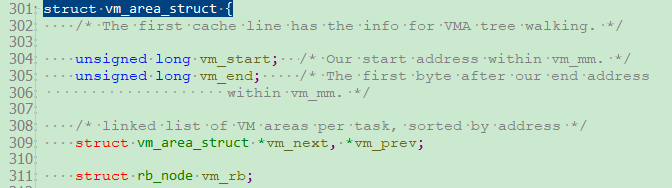
Section header string table index，段表字符串表所在的段在段表中的下标。本例中等于10，见图 7。

进程地址空间

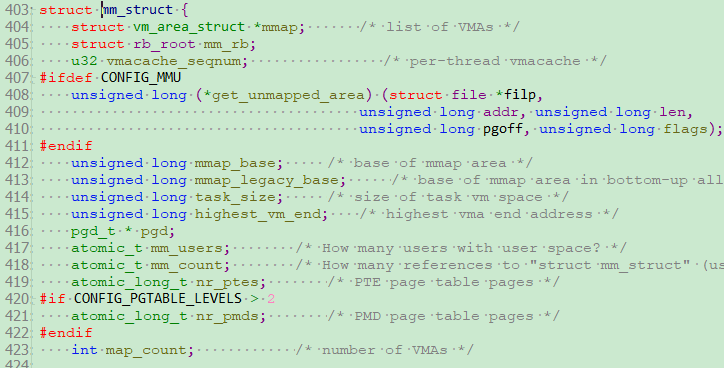
进程可寻址的虚拟地址空间

结构：struct vm\_area\_struct 数据结构描述vma进程地址空间或进程线性区

【\include\linux\mm\_types.h】



内存描述符：struct mm\_struct {



static int kmem\_cache\_open(struct kmem\_cache \*s, unsigned long flags)

【\mm\slub.c】

int \_\_kmem\_cache\_create(struct kmem\_cache \*s, unsigned long flags)

static int kmem\_cache\_open(struct kmem\_cache \*s, unsigned long flags)

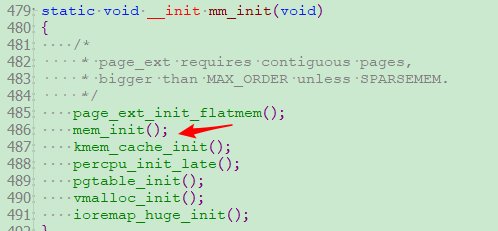
static inline int alloc\_kmem\_cache\_cpus(struct kmem\_cache \*s)

static void init\_kmem\_cache\_cpus(struct kmem\_cache \*s)

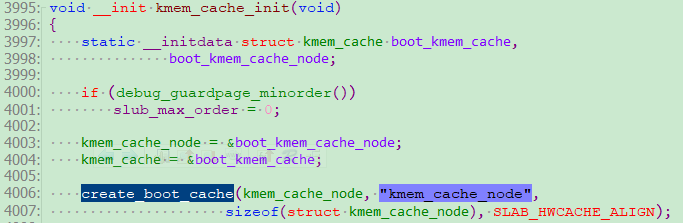
mm\_init(void)

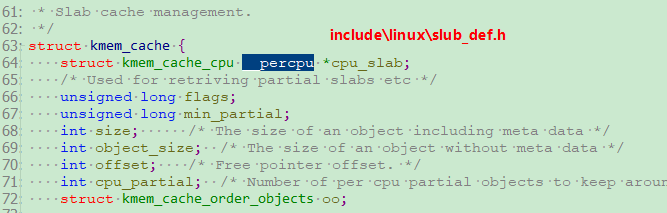
初始化部分

[\init\main.c] static void \_\_init mm\_init(void)



void \_\_init kmem\_cache\_init(void) [\mm\slub.c]





**【物理内存】**

物理页面：内存管理最小粒度：

* 32位的处理器寻址时按照数据位宽，也就，但是处理在处理物理内存时却不是按照字节来分配的。 因为现在的处理都是采用分面机制来管理内存。
* Linux 内核中使用一个struct page 数据结构来描述一个物理页表。

Page 数据结构重要成员解析

* \_count 引用计数
  + \_count表示内核中引用该页面的次数。当\_count的值为0时，表示该page页面为空闲或者即将被释放的页面。当\_count的值大于0时，表示该page页面已经被分配且内核正使用。暂时不会被释放。
  + 内核中常用的加减\_count引用计数API为

get\_page,put\_page,page\_cache\_get,page\_cache\_release

* \_mapcount引用计数
  + 表示这个面面被进程映射的个数，即已经映射多少个用户pte页表。\_mapcount引用计数主要用于RMAP反射映射机制中。
  + 内核提供的两个宏来统计某个面面的\_count 和\_mapcount的计数。

**数据结构struct page 大小**

* 每个物理页面一个struct page 数据结构，若数据结构本身很大，会很浪费内存。
* 一个物理页面是4096 字节，假设struct page 数据结构占用40字节。那么就相当于要浪费百分之一内存来存放struct page 数据结构

**内存管理zone**

为什么要有zone?

* 32位机器 ，在4GB的地址空间中，通常内核空间只有1GB大小，那么对于大小为4GB物理内存无法进行 一一线性映射的。
* Linux 内核的做法把物理内存分成两部分，其中一部分是线性映射的。如果用一个内存管理 zone 的概念来描述它，那就是zone\_normal区，剩余部分叫作higth memory ,也同样使用一个内存管理区来描述它，称为zone\_highmem

Zone都有哪些?

* Zone DMA
* ZONE\_NORMAL :用与线性映射内存
* ZONE\_HIGHMEM: 用于管理高端内存，这些高端内存是不能线性映射到内核地址空间。注意在64位的处理川口春奈不需要这个管理区

Linux 操作系统和驱动程序运行在内核空间，应用程序运行在用户空间，两者不能简单地使用指针传递数据，因为Linux使用的虚拟内存机制，用户空间的数据可能被换出，当内核空间使用用户空间指针时，对应的数据可能不在内存中。

**Linux内核地址空间划分**

通常**32位Linux内核**地址空间划分0~3G为用户空间，3~4G为内核空间。注意这里是32位内核地址空间划分，64位内核地址空间划分是不同的。

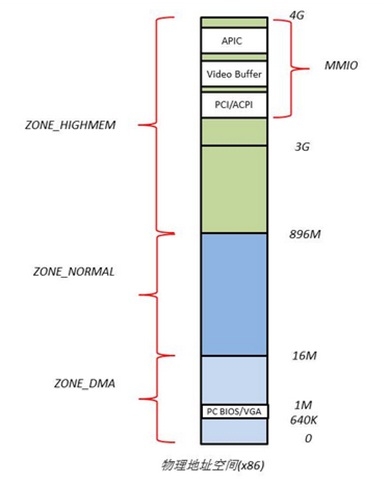
**1、x86的物理地址空间布局：**

物理地址空间的顶部以下一段空间，被PCI设备的I/O内存映射占据，它们的大小和布局由PCI规范所决定。640K~1M这段地址空间被BIOS和VGA适配器所占据。

Linux系统在初始化时，会根据实际的物理内存的大小，为每个物理页面创建一个page对象，所有的page对象构成一个mem\_map数组。

进一步，针对不同的用途，Linux内核将所有的物理页面划分到3类内存管理区中，如图，分别为ZONE\_DMA，ZONE\_NORMAL，ZONE\_HIGHMEM。

ZONE\_DMA的范围是0~16M，该区域的物理页面专门供I/O设备的DMA使用。之所以需要单独管理DMA的物理页面，是因为DMA使用物理地址访问内存，不经过MMU，并且需要连续的缓冲区，所以为了能够提供物理上连续的缓冲区，必须从物理地址空间专门划分一段区域用于DMA。

[](http://s3.51cto.com/wyfs02/M01/25/CF/wKiom1Nm_tiyhQPXAACWQfYUvVU353.jpg)

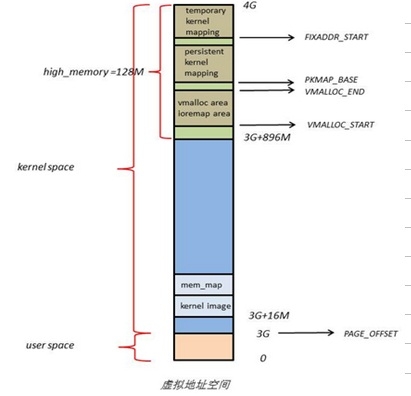
ZONE\_NORMAL的范围是16M~896M，该区域的物理页面是内核能够直接使用的。

ZONE\_HIGHMEM的范围是896M~结束，该区域即为高端内存，内核不能直接使用。

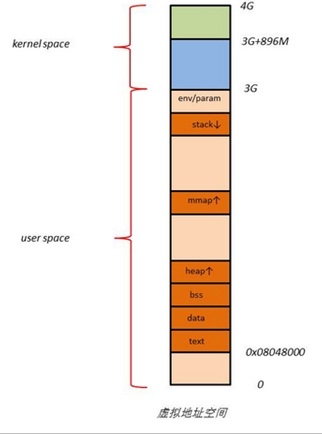
**2、linux虚拟地址内核空间分布**

 在kernel image下面有16M的内核空间用于DMA操作。位于内核空间高端的128M地址主要由3部分组成，分别为vmalloc area，持久化内核映射区，临时内核映射区。

由于ZONE\_NORMAL和内核线性空间存在直接映射关系，所以内核会将频繁使用的数据如kernel代码、GDT、IDT、PGD、mem\_map数组等放在ZONE\_NORMAL里。而将用户数据、页表(PT)等不常用数据放在ZONE\_ HIGHMEM里，只在要访问这些数据时才建立映射关系(kmap())。比如，当内核要访问I/O设备存储空间时，就使用ioremap()将位于物理地址高端的mmio区内存映射到内核空间的vmalloc area中，在使用完之后便断开映射关系。

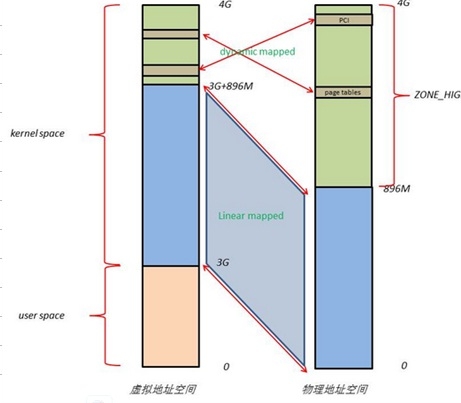
[](http://s3.51cto.com/wyfs02/M02/25/CF/wKiom1Nm_tezL0_TAACljE7bXGU278.jpg)

**3、linux虚拟地址用户空间分布**

**[](http://s3.51cto.com/wyfs02/M00/25/CF/wKiom1NnADvC07kmAACCVbbosf0493.jpg)**

用户进程的代码区一般从虚拟地址空间的0x08048000开始，这是为了便于检查空指针。代码区之上便是数据区，未初始化数据区，堆区，栈区，以及参数、全局环境变量。

**4、linux虚拟地址与物理地址映射的关系**

[](http://s3.51cto.com/wyfs02/M02/25/D0/wKioL1Nm_qiCvmYxAADRFZV_1jQ205.jpg)

 Linux将4G的线性地址空间分为2部分，0~3G为user space，3G~4G为kernel space。

由于开启了分页机制，内核想要访问物理地址空间的话，必须先建立映射关系，然后通过虚拟地址来访问。为了能够访问所有的物理地址空间，就要将全部物理地址空间映射到1G的内核线性空间中，这显然不可能。于是，内核将0~896M的物理地址空间一对一映射到自己的线性地址空间中，这样它便可以随时访问ZONE\_DMA和ZONE\_NORMAL里的物理页面；此时内核剩下的128M线性地址空间不足以完全映射所有的ZONE\_HIGHMEM，Linux采取了动态映射的方法，即按需的将ZONE\_HIGHMEM里的物理页面映射到kernel space的最后128M线性地址空间里，使用完之后释放映射关系，以供其它物理页面映射。虽然这样存在效率的问题，但是内核毕竟可以正常的访问所有的物理地址空间了。

**5、buddyinfo的理解**

cat /proc/buddyinfo 显示如下：

Node 0, zone      DMA       0      4      5      4      4      3 ...

Node 0, zone   Normal      1      0      0      1    101     8 ...

Node 0, zone  HighMem    2      0      0      1      1       0 ...

其中，Node表示在NUMA环境下的节点号，这里只有一个节点0；zone表示每一个节点下的区域，一般有DMA、Normal和HignMem三个区域；后面的列表示，伙伴系统中每一个order对应的空闲页面块。例如，对于zone DMA的第二列（从0开始算起），空闲页面数为5\*2^4，可用内存为5\*2^4\*PAGE\_SIZE。

计算方法就是：

当前列的数字\*2^列数\*PAGE\_SIZE 其中列数是从0开始计算的，即第一列是 当前列的数字\*2^0\*PAGE\_SIZE

常见问题：

**1、用户空间（进程）是否有高端内存概念？**

用户进程没有高端内存概念。只有在内核空间才存在高端内存。用户进程最多只可以访问3G物理内存，而内核进程可以访问所有物理内存。

**2、64位内核中有高端内存吗？**

目前现实中，64位Linux内核不存在高端内存，因为64位内核可以支持超过512GB内存。若机器安装的物理内存超过内核地址空间范围，就会存在高端内存。

**3、用户进程能访问多少物理内存？内核代码能访问多少物理内存？**

32位系统用户进程最大可以访问3GB，内核代码可以访问所有物理内存。

64位系统用户进程最大可以访问超过512GB，内核代码可以访问所有物理内存。

**4、高端内存和物理地址、逻辑地址、线性地址的关系？**

高端内存只和逻辑地址有关系，和逻辑地址、物理地址没有直接关系。

**5、为什么不把所有的地址空间都分配给内核？**

若把所有地址空间都给内存，那么用户进程怎么使用内存？怎么保证内核使用内存和用户进程不起冲突？

**linux NUMA技术**

1. 概述

NUMA（Non-Uniform Memory Access Architecture）即非一致性内存访问技术。NUMA系统有多个Node通过高速互连的网络联系起来的系统。而Node则是由一组cpu和本地内存组成。不同的Node有不同的物理内存，由于Node访问本地内存和访问其它节点的内存的速度是不一致的，为了解决非一致性访问内存对性能的影响，有一些工具可以使用。包括 numactl，autonuma， HPE-AIX等。

分配：slub debug模式