# Linux内核模块机制

## 一. 什么是内核模块？内核模块的优缺点？

内核模块是Linux内核向外部提供的一个插口，其全称为动态可加载内核模块（Loadable Kernel Module，LKM），我们简称为模块。Linux内核之所以提供模块机制，是因为它本身是一个单内核（monolithic kernel）。单内核的最大优点是效率高，因为所有的内容都集成在一起，但其缺点是可扩展性和可维护性相对较差，模块机制就是为了弥补这一缺陷。

模块是具有独立功能的程序，它可以被单独编译，但不能独立运行。它在运行时被链接到内核作为内核的一部分在内核空间运行，这与运行在用户空间的进程是不同的。模块通常由一组函数和数据结构组成，用来实现一种文件系统、一个驱动程序或其他内核上层的功能。

总之，模块是一个为内核（从某种意义上来说，内核也是一个模块）或其他内核模块提供使用功能的代码块。

利用内核模块的动态装载性具有如下优点：

1. 将内核映象的尺寸保持在最小，并具有最大的灵活性；
2. 便于检验新的内核代码，而不需重新编译内核并重新引导。

但是，内核模块的引入也带来了如下问题：

1）对系统性能和内存利用有负面影响；

2）装入的内核模块和其他内核部分一样，具有相同的访问权限，因此，差的内核模块 会导致系统崩溃；

3）为了使内核模块访问所有内核资源，内核必须维护符号表，并在装入和卸载模块时 修改这些符号表；

4）有些模块要求利用其他模块的功能，因此，内核要维护模块之间的依赖性。

5） 内核必须能够在卸载模块时通知模块，并且要释放分配给模块的内存和中断等资源；

6）内核版本和模块版本的不兼容，也可能导致系统崩溃，因此，严格的版本检查是必需的。

尽管内核模块的引入同时也带来不少问题，但是模块机制确实是扩充内核功能一种行之有效的方法，也是在内核级进行编程的有效途径。

## 二. 模块的安装和卸载？

### 1.模块的安装

在Linux中，对模块进行加载有两种方法，一种是手工加载，另一种是自动加载。前者使用insmod或者modprobe命令来实现，而后者通过守护进程kerneld(2.0.x内核)或者内核线程kmod(2.1.x以后的内核) 来实现，不过不管是kerneld，还是kmod最终都调用了modprobe来实现模块的加载。这里以insmod为例来分析模块的加载过程。

insmod首先通过系统调用query\_module()遍历模块链表来获得系统中的所有符号及其在内存中的物理地址。然后利用得到的符号表修正模块中引用到的外部符号，在此过程中记录下该模块所要用到的模块。由于是直接用这些符号在内存中的物理地址进行更正的，所以模块对内核空间的地址引用是正确的。如果此时该模块还有一些符号的地址未知，则该模块不能被加载。

然后insmod填写模块module\_ref表，其中dep指向本模块所使用的模块，即在修正本模块使用到的外部符号过程中标记过的模块，这些模块在内存中的物理地址也是通过系统调用query\_module得到的。而ref指针则留待系统以后填写，next\_ref设为NULL。

由于别的模块可能要使用本模块提供的服务，所以还需要提供本模块的输出符号表，这一工作也由insmod来完成。

接着insmod发出系统调用create\_module()，由该系统调用为模块分配足够的内核空间，并初始化位于该空间起始处的struct module结构。然后insmod通过系统调用init\_module()让系统完成余下的工作。init\_module的主要工作有三个：1.将模块映像从用户空间复制到内核空间；2.更新该模块使用到的所有已装入模块的引用链表；3.调用该模块的初始化函数init\_module。一旦模块载入内核后，则成为内核代码的一部分，与其它内核代码地位是相同的。这样，insmod就完成了将模块加载到系统中的功能。

### 2.模块的卸载

相对于模块的安装而言，模块的卸载是比较简单的。与模块的安装相对应，模块的卸载也有两种方法：第一种是用户使用rmmod命令卸载module，第二种是使用kerneld或者kmod自动卸载。

就rmmod而言，它在找出要卸载的模块后，根据其引用链表检查是否有别的模块要使用本模块。若有，则打印出错信息并终止；否则发出系统调用delete\_module()。delete\_module将调用free\_module,由free\_module依次完成下面四项工作：1.调用该模块的cleanup子程序，以释放系统分配给该模块的资源；2.修改该模块所依赖的所有模块的引用链表，将要卸载的模块从它们的引用链表中删除；3.将该模块从系统的模块链表中删除；4.释放分配给该模块的核心内存。

## 三.模块的内存分配

1.原理说明  
Linux内核中采用了一种同时适用于32位和64位系统的内存分页模型，对于32位系统来说，两级页表足够用了，而在x86\_64系统中，用到了四级页表，四级页表分别为：

         页全局目录(Page Global Directory)

         页上级目录(Page Upper Directory)

         页中间目录(Page Middle Directory)

         页表(Page Table)

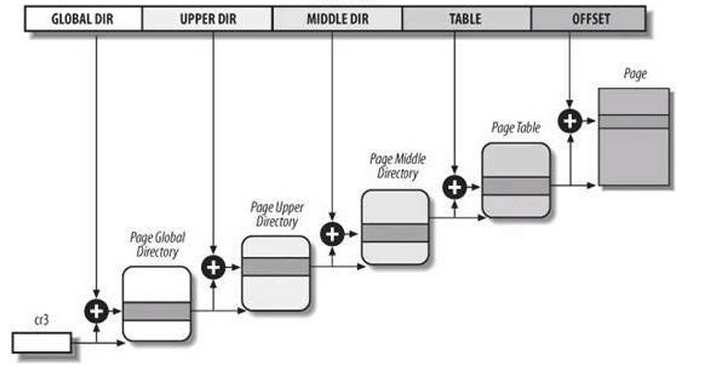


图2

    页全局目录包含若干页上级目录的地址，页上级目录又依次包含若干页中间目录的地址，而页中间目录又包含若干页表的地址，每一个页表项指向一个页框。Linux中采用4KB大小的页框作为标准的内存分配单元。

多级分页目录结构

1.1.伙伴系统算法

    在实际应用中，经常需要分配一组连续的页框，而频繁地申请和释放不同大小的连续页框，必然导致在已分配页框的内存块中分散了许多小块的空闲页框。这样，即使这些页框是空闲的，其他需要分配连续页框的应用也很难得到满足。

    为了避免出现这种情况，Linux内核中引入了伙伴系统算法(buddy system)。把所有的空闲页框分组为11个块链表，每个块链表分别包含大小为1，2，4，8，16，32，64，128，256，512和1024个连续页框的页框块。最大可以申请1024个连续页框，对应4MB大小的连续内存。每个页框块的第一个页框的物理地址是该块大小的整数倍。

    假设要申请一个256个页框的块，先从256个页框的链表中查找空闲块，如果没有，就去512个页框的链表中找，找到了则将页框块分为2个256个页框的块，一个分配给应用，另外一个移到256个页框的链表中。如果512个页框的链表中仍没有空闲块，继续向1024个页框的链表查找，如果仍然没有，则返回错误。

    页框块在释放时，会主动将两个连续的页框块合并为一个较大的页框块。1.2. slab分配器

    slab分配器源于 Solaris 2.4 的分配算法，工作于物理内存页框分配器之上，管理特定大小对象的缓存，进行快速而高效的内存分配。

    slab分配器为每种使用的内核对象建立单独的缓冲区。Linux 内核已经采用了伙伴系统管理物理内存页框，因此 slab分配器直接工作于伙伴系统之上。每种缓冲区由多个 slab 组成，每个 slab就是一组连续的物理内存页框，被划分成了固定数目的对象。根据对象大小的不同，缺省情况下一个 slab 最多可以由 1024个页框构成。出于对齐等其它方面的要求，slab 中分配给对象的内存可能大于用户要求的对象实际大小，这会造成一定的内存浪费。

2.常用内存分配函数

2.1.  \_\_get\_free\_pages

    unsigned long \_\_get\_free\_pages(gfp\_t gfp\_mask, unsigned int order)  
    \_\_get\_free\_pages函数是最原始的内存分配方式，直接从伙伴系统中获取原始页框，返回值为第一个页框的起始地址。\_\_get\_free\_pages在实现上只是封装了alloc\_pages函数，从代码分析，alloc\_pages函数会分配长度为1<<order的连续页框块。order参数的最大值由include/linux/Mmzone.h文件中的MAX\_ORDER宏决定，在默认的2.6.18内核版本中，该宏定义为10。也就是说在理论上\_\_get\_free\_pages函数一次最多能申请1<<10 \* 4KB也就是4MB的连续物理内存。但是在实际应用中，很可能因为不存在这么大量的连续空闲页框而导致分配失败。在测试中，order为10时分配成功，order为11则返回错误。

2.2. kmem\_cache\_alloc

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

        size\_t align, unsigned long flags,  
        void (\*ctor)(void\*, struct kmem\_cache \*, unsigned long),  
  
    void \*kmem\_cache\_alloc(struct kmem\_cache \*c, gfp\_t flags)  
    kmem\_cache\_create/ kmem\_cache\_alloc是基于slab分配器的一种内存分配方式，适用于反复分配释放同一大小内存块的场合（可以小于页大小）。首先用kmem\_cache\_create创建一个高速缓存区域，然后用kmem\_cache\_alloc从该高速缓存区域中获取新的内存块。 kmem\_cache\_alloc一次能分配的最大内存由mm/slab.c文件中的MAX\_OBJ\_ORDER宏定义，在默认的2.6.18内核版本中，该宏定义为5，于是一次最多能申请1<<5 \* 4KB也就是128KB的连续物理内存。分析内核源码发现，kmem\_cache\_create函数的size参数大于128KB时会调用BUG()。测试结果验证了分析结果，用kmem\_cache\_create分配超过128KB的内存时使内核崩溃。  
2.3. mempool\_alloc  
   void \*mempool\_alloc(mempool\_t \*pool,int gfp\_mask)  
   为了确保在内存分配不允许失败情况下成功分配内存，内核提供了称为内存池( "mempool" )的抽象，它其实是某种后备高速缓存,mempool的底层通常使用slab。它为了紧急情况下的使用。所以使用时必须注意：mempool会分配一些内存块，使其空闲而不真正使用，所以容易消耗大量内存。而且不要使用mempool处理可失败的分配。应避免在驱动代码中使用mempool。  
2.4.  kmalloc  
    void \*kmalloc(size\_t size, gfp\_t flags)  
    kmalloc是内核中最常用的一种内存分配方式，它通过调用kmem\_cache\_alloc函数来实现。kmalloc一次最多能申请的内存大小由include/linux/Kmalloc\_size.h的内容来决定，在默认的2.6.18内核版本中，kmalloc一次最多能申请大小为131702B也就是128KB字节的连续物理内存。测试结果表明，如果试图用kmalloc函数分配大于128KB的内存，编译不能通过。  
2.5.  vmalloc  
    void \*vmalloc(unsigned long size)  
    前面几种内存分配方式都是物理连续的，能保证较低的平均访问时间。但是在某些场合中，对内存区的请求不是很频繁，较高的内存访问时间也可以接受，这是就可以分配一段线性连续，物理不连续的地址，带来的好处是一次可以分配较大块的内存。图3-1表示的是vmalloc分配的内存使用的地址范围。vmalloc对一次能分配的内存大小没有明确限制。出于性能考虑，应谨慎使用vmalloc函数。在测试过程中，最大能一次分配1GB的空间。  
2.6.  dma\_alloc\_coherent  
    void \*dma\_alloc\_coherent(struct device \*dev, size\_t size, ma\_addr\_t \*dma\_handle, gfp\_t gfp)  
    DMA是一种硬件机制，允许外围设备和主存之间直接传输IO数据，而不需要CPU的参与，使用DMA机制能大幅提高与设备通信的吞吐量。DMA操作中，涉及到CPU高速缓存和对应的内存数据一致性的问题，必须保证两者的数据一致，在x86\_64体系结构中，硬件已经很好的解决了这个问题， dma\_alloc\_coherent和\_\_get\_free\_pages函数实现差别不大，前者实际是调用\_\_alloc\_pages函数来分配内存，因此一次分配内存的大小限制和后者一样。\_\_get\_free\_pages分配的内存同样可以用于DMA操作。测试结果证明，dma\_alloc\_coherent函数一次能分配的最大内存也为4M。  
2.7.  ioremap  
    void \* ioremap (unsigned long offset, unsigned long size)  
    ioremap是一种更直接的内存“分配”方式，使用时直接指定物理起始地址和需要分配内存的大小，然后将该段物理地址映射到内核地址空间。ioremap用到的物理地址空间都是事先确定的，和上面的几种内存分配方式并不太一样，并不是分配一段新的物理内存。ioremap多用于设备驱动，可以让CPU直接访问外部设备的IO空间。ioremap能映射的内存由原有的物理内存空间决定，所以没有进行测试。  
2.8. Boot Memory  
    如果要分配大量的连续物理内存，上述的分配函数都不能满足，就只能用比较特殊的方式，在Linux内核引导阶段来预留部分内存。  
2.8.1. 在内核引导时分配内存  
    void\* alloc\_bootmem(unsigned long size)  
    可以在Linux内核引导过程中绕过伙伴系统来分配大块内存。使用方法是在Linux内核引导时，调用mem\_init函数之前用alloc\_bootmem函数申请指定大小的内存。如果需要在其他地方调用这块内存，可以将alloc\_bootmem返回的内存首地址通过EXPORT\_SYMBOL导出，然后就可以使用这块内存了。这种内存分配方式的缺点是，申请内存的代码必须在链接到内核中的代码里才能使用，因此必须重新编译内核，而且内存管理系统看不到这部分内存，需要用户自行管理。测试结果表明，重新编译内核后重启，能够访问引导时分配的内存块。  
2.8.2. 通过内核引导参数预留顶部内存  
    在Linux内核引导时，传入参数“mem＝size”保留顶部的内存区间。比如系统有256MB内存，参数“mem＝248M”会预留顶部的8MB内存，进入系统后可以调用ioremap(0xF800000，0x800000)来申请这段内存。

3.几种分配函数的比较

|  |  |  |  |
| --- | --- | --- | --- |
|  | 分配原理 | 最大内存 | 其他 |
| **\_\_get\_free\_pages** | 直接对页框进行操作 | 4MB | 适用于分配较大量的连续物理内存 |
| **kmem\_cache\_alloc** | 基于slab机制实现 | 128KB | 适合需要频繁申请释放相同大小内存块时使用 |
| **kmalloc** | 基于kmem\_cache\_alloc实现 | 128KB | 最常见的分配方式，需要小于页框大小的内存时可以使用 |
| **vmalloc** | 建立非连续物理内存到虚拟地址的映射 |  | 物理不连续，适合需要大内存，但是对地址连续性没有要求的场合 |
| **dma\_alloc\_coherent** | 基于\_\_alloc\_pages实现 | 4MB | 适用于DMA操作 |
| **ioremap** | 实现已知物理地址到虚拟地址的映射 |  | 适用于物理地址已知的场合，如设备驱动 |
| **alloc\_bootmem** | 在启动kernel时，预留一段内存，内核看不见 |  | 小于物理内存大小，内存管理要求较高 |

   注：表中提到的最大内存数据来自CentOS5.3 x86\_64系统，其他系统和体系结构会有不同

参考文献

[1] 孙海彬. 傅谦. 许良贤. Linux内核模块实现机制， 2000,3

[2]百度百科：Linux内核模块机制http://baike.baidu.com/link?url=dpIKhDa3w4ebmi6IYM8J7AqtZs\_ceVTGedoox9UVi0IQEmIXa5QyZL92pxQnCjzyREP1PR4I3KUjqbP4BV6o4a