# [内核简介](http://www.cnblogs.com/wang_yb/archive/2012/08/15/2640972.html)

本篇简单介绍内核相关的基本概念。

**主要内容：**

* 单内核和微内核
* 内核版本号

## ****单内核和微内核****

|  |  |  |  |
| --- | --- | --- | --- |
|  | **原理** | **优势** | **劣势** |
| **单内核** | 整个内核都在一个大内核地址空间上运行。 | 1. 简单。 2. 高效：所有内核都在一个大的地址空间上，所以内核各个功能之间的调用和调用函数类似，几乎没有性能开销。 | 一个功能的崩溃会导致整个内核无法使用。 |
| **微内核** | 内核按功能被划分成各个独立的过程。每个过程独立的运行在自己的地址空间上。 | 1. 安全：内核的各种服务独立运行，一种服务挂了不会影响其他服务。 | 内核各个服务之间的调用涉及进程间的通信，比较复杂且效率低。 |

Linux的内核虽然是基于单内核的，但是经过这么多年的发展，也具备微内核的一些特征。（体现了Linux实用至上的原则）

主要有以下特征：

支持动态加载内核模块

支持对称多处理（SMP）

内核可以抢占（preemptive），允许内核运行的任务有优先执行的能力

不区分线程和进程

## ****内核版本号****

内核的版本号主要有四个数组组成。比如版本号：2.6.26.1  其中，

2  - 主版本号

6  - 从版本号或副版本号

26 - 修订版本号

1  - 稳定版本号

副版本号表示这个版本是稳定版（**偶数**）还是开发版（**奇数**），上面例子中的版本号是稳定版。

稳定的版本可用于企业级环境。

修订版本号的升级包括BUG修正，新的驱动以及新的特性的追加。

稳定版本号主要是一些关键性BUG的修改。

# [内核开发的准备](http://www.cnblogs.com/wang_yb/archive/2012/08/16/2641836.html)

在尝试内核开发之前，需要对内核有个整体的了解。

**主要内容：**

* 获取内核源码
* 内核源码的结构
* 编译内核的方法
* 内核开发的特点

## ****获取内核源码****

内核是开源的，所有获取源码特别方便，参照以下的网址，可以通过git或者直接下载压缩好的源码包。

[http://www.kernel.org](http://www.kernel.org/)

## ****内核源码的结构****

|  |  |
| --- | --- |
| **目录** | **说明** |
| arch | 特定体系结构的代码 |
| block | 块设备I/O层 |
| crypo | 加密API |
| Documentation | 内核源码文档 |
| drivers | 设备驱动程序 |
| firmware | 使用某些驱动程序而需要的设备固件 |
| fs | VFS和各种文件系统 |
| include | 内核头文件 |
| init | 内核引导和初始化 |
| ipc | 进程间通信代码 |
| kernel | 像调度程序这样的核心子系统 |
| lib | 同样内核函数 |
| mm | 内存管理子系统和VM |
| net | 网络子系统 |
| samples | 示例，示范代码 |
| scripts | 编译内核所用的脚本 |
| security | Linux 安全模块 |
| sound | 语音子系统 |
| usr | 早期用户空间代码（所谓的initramfs） |
| tools | 在Linux开发中有用的工具 |
| virt | 虚拟化基础结构 |

## ****编译内核的方法****

还未实际尝试过手动编译内核，只是用yum更新过内核。这部分等以后手动编译过再补上。

安装新的内核后，重启时会提示进入哪个内核。当多次安装新的内核后，启动列表会很长（因为有很多版本的内核），显得不是很方便。

下面介绍3种删除那些不用的内核的方法：(是如何安装的就选择相应的删除方法)

#### 3.1 rpm 删除法

   rpm -qa | grep kernel\*  (查找所有linux内核版本)  
   rpm -e kernel-(想要删除的版本)

#### 3.2 yum 删除法

   yum remove kernel-(要删除的版本)

#### 3.3 手动删除

   删除/lib/modules/目录下不需要的内核库文件  
   删除/usr/src/kernel/目录下不需要的内核源码  
   删除/boot目录下启动的核心档案禾内核映像  
   更改grub的配置，删除不需要的内核启动列表

## ****内核开发的特点****

#### 4.1  无标准C库

为了保证内核的小和高效，内核开发中不能使用C标准库，所以连最常用的printf函数也没有，但是还好有个printk函数来代替。

#### 4.2 使用GNU C，推荐用gcc 4.4或以后的版本来编译内核

因为使用GNU C，所有内核中常使用GNU C中的一些扩展：

##### 4.2.1 内联函数

内联函数在编译时会在它被调用的地方展开，减少了函数调用的开销，性能较好。但是，频繁的使用内联函数也会使代码变长，从而在运行时占用更多的内存。

所以内联函数使用时最好要满足以下几点：函数较小，会被反复调用，对程序的时间要求比较严格。

内联函数示例：static **inline** void sample();

##### 4.2.2 内联汇编

内联汇编用于偏近底层或对执行时间严格要求的地方。示例如下：

unsigned int low, high;

asm volatile("rdtsc" : "=a" (low), "=d" (high));

/\* low 和 high 分别包含64位时间戳的低32位和高32位 \*/

##### 4.2.3 分支声明

如果能事先判断一个if语句时经常为真还是经常为假，那么可以用unlikely和likely来优化这段判断的代码。

[复制代码](javascript:void(0);)

/\* 如果error在绝大多数情况下为0(假) \*/

if (unlikely(error)) {

/\* ... \*/

}

/\* 如果success在绝大多数情况下不为0(真) \*/

if (likely(success)) {

/\* ... \*/

}

[复制代码](javascript:void(0);)

#### 4.3 没有内存保护

因为内核是最低层的程序，所以如果内核访问的非法内存，那么整个系统都会挂掉！！所以内核开发的风险比用户程序开发的风险要大。

而且，内核中的内存是不分页的，每用一个字节的内存，物理内存就少一个字节。所以内核中使用内存一定要谨慎。

#### 4.4 不使用浮点数

内核不能完美的支持浮点操作，使用浮点数时，需要人工保存和恢复浮点寄存器及其他一些繁琐的操作。

#### 4.5 内核栈容积小且固定

内核栈的大小有编译内核时决定的，对于不用的体系结构，内核栈的大小虽然不一样，但都是固定的。

查看内核栈大小的方法：

ulimit -a | grep "stack size"

#### 4.6 同步和并发

Linux是多用户的操作系统，所以必须处理好同步和并发操作，防止因竞争而出现死锁。

#### 4.7 可移植性

Linux内核可用于不用的体现结构，支持多种硬件。所以开发时要时刻注意可移植性，尽量使用体系结构无关的代码

# [Linux的进程](http://www.cnblogs.com/wang_yb/archive/2012/08/20/2647912.html)

进程是所有操作系统的核心概念，同样在linux上也不例外。

**主要内容：**

* 进程和线程
* 进程的生命周期
* 进程的创建
* 进程的终止

### ****1. 进程和线程****

进程和线程是程序运行时状态，是动态变化的，进程和线程的管理操作(比如，创建，销毁等)都是有内核来实现的。

Linux中的进程于Windows相比是很轻量级的，而且不严格区分进程和线程，线程不过是一种特殊的进程。

所以下面只讨论进程，只有当线程与进程存在不一样的地方时才提一下线程。

进程提供2种虚拟机制：虚拟处理器和虚拟内存

每个进程有独立的虚拟处理器和虚拟内存，

每个线程有独立的虚拟处理器，同一个进程内的线程有可能会共享虚拟内存。

内核中进程的信息主要保存在task\_struct中(include/linux/sched.h)

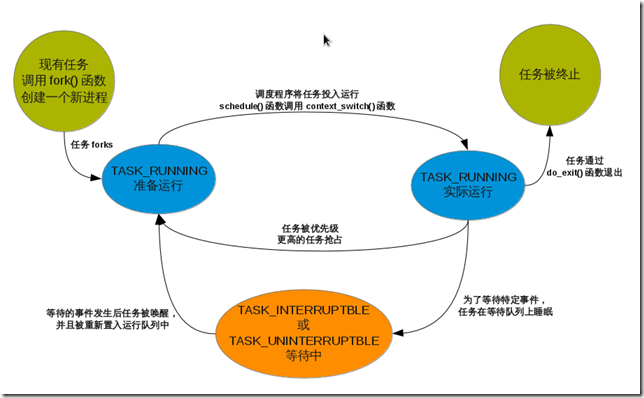
进程标识PID和线程标识TID对于同一个进程或线程来说都是相等的。

Linux中可以用ps命令查看所有进程的信息：

ps -eo pid,tid,ppid,comm

### ****2. 进程的生命周期****

进程的各个状态之间的转化构成了进程的整个生命周期。

[](http://images.cnblogs.com/cnblogs_com/wang_yb/201208/201208201741296349.png)

### ****3. 进程的创建****

Linux中创建进程与其他系统有个主要区别，Linux中创建进程分2步：fork()和exec()。

fork: 通过拷贝当前进程创建一个子进程

exec: 读取可执行文件，将其载入到内存中运行

创建的流程：

1. 调用dup\_task\_struct()为新进程分配内核栈，task\_struct等，其中的内容与父进程相同。
2. check新进程(进程数目是否超出上限等)
3. 清理新进程的信息(比如PID置0等)，使之与父进程区别开。
4. 新进程状态置为 TASK\_UNINTERRUPTIBLE
5. 更新task\_struct的flags成员。
6. 调用alloc\_pid()为新进程分配一个有效的PID
7. 根据clone()的参数标志，拷贝或共享相应的信息
8. 做一些扫尾工作并返回新进程指针

创建进程的fork()函数实际上最终是调用clone()函数。

创建线程和进程的步骤一样，只是最终传给clone()函数的参数不同。

比如，通过一个普通的fork来创建进程，相当于：clone(SIGCHLD, 0)

创建一个和父进程共享地址空间，文件系统资源，文件描述符和信号处理程序的进程，即一个线程：clone(CLONE\_VM | CLONE\_FS | CLONE\_FILES | CLONE\_SIGHAND, 0)

在内核中创建的内核线程与普通的进程之间还有个主要区别在于：内核线程没有独立的地址空间，它们只能在内核空间运行。

这与之前提到的Linux内核是个单内核有关。

### ****4. 进程的终止****

和创建进程一样，终结一个进程同样有很多步骤：

子进程上的操作(do\_exit)

1. 设置task\_struct中的标识成员设置为PF\_EXITING
2. 调用del\_timer\_sync()删除内核定时器, 确保没有定时器在排队和运行
3. 调用exit\_mm()释放进程占用的mm\_struct
4. 调用sem\_\_exit()，使进程离开等待IPC信号的队列
5. 调用exit\_files()和exit\_fs()，释放进程占用的文件描述符和文件系统资源
6. 把task\_struct的exit\_code设置为进程的返回值
7. 调用exit\_notify()向父进程发送信号，并把自己的状态设为EXIT\_ZOMBIE
8. 切换到新进程继续执行

子进程进入EXIT\_ZOMBIE之后，虽然永远不会被调度，关联的资源也释放掉了，但是它本身占用的内存还没有释放，  
比如创建时分配的内核栈，task\_struct结构等。这些由父进程来释放。

父进程上的操作(release\_task)

父进程受到子进程发送的exit\_notify()信号后，将该子进程的进程描述符和所有进程独享的资源全部删除。

从上面的步骤可以看出，必须要确保每个子进程都有父进程，如果父进程在子进程结束之前就已经结束了会怎么样呢？

子进程在调用exit\_notify()时已经考虑到了这点。

如果子进程的父进程已经退出了，那么子进程在退出时，exit\_notify()函数会先调用forget\_original\_parent()，然后再调用find\_new\_reaper()来寻找新的父进程。

find\_new\_reaper()函数先在当前线程组中找一个线程作为父亲，如果找不到，就让init做父进程。(init进程是在linux启动时就一直存在的)

# [进程的调度](http://www.cnblogs.com/wang_yb/archive/2012/09/04/2670564.html)

**主要内容：**

* 什么是调度
* 调度实现原理
* Linux上调度实现的方法
* 调度相关的系统调用

### ****1. 什么是调度****

现在的操作系统都是多任务的，为了能让更多的任务能同时在系统上更好的运行，需要一个管理程序来管理计算机上同时运行的各个任务（也就是进程）。

这个管理程序就是调度程序，它的功能说起来很简单：

1. 决定哪些进程运行，哪些进程等待
2. 决定每个进程运行多长时间

此外，为了获得更好的用户体验，运行中的进程还可以立即被其他更紧急的进程打断。

总之，调度是一个平衡的过程。一方面，它要保证各个运行的进程能够最大限度的使用CPU(即尽量少的切换进程，进程切换过多，CPU的时间会浪费在切换上)；另一方面，保证各个进程能公平的使用CPU(即防止一个进程长时间独占CPU的情况)。

### ****2. 调度实现原理****

前面说过，调度功能就是决定哪个进程运行以及进程运行多长时间。

决定哪个进程运行以及运行多长时间都和进程的优先级有关。为了确定一个进程到底能持续运行多长时间，调度中还引入了时间片的概念。

#### 2.1 关于进程的优先级

进程的优先级有2种度量方法，一种是nice值，一种是实时优先级。

nice值的范围是-20～+19，值越大优先级越低，也就是说nice值为-20的进程优先级最大。

实时优先级的范围是0～99，与nice值的定义相反，实时优先级是值越大优先级越高。

实时进程都是一些对响应时间要求比较高的进程，因此系统中有实时优先级高的进程处于运行队列的话，它们会抢占一般的进程的运行时间。

进程的2种优先级会让人不好理解，到底哪个优先级更优先？一个进程同时有2种优先级怎么办？

其实linux的内核早就有了解决办法。

对于第一个问题，到底哪个优先级更优先？

答案是实时优先级高于nice值，在内核中，实时优先级的范围是 0～MAX\_RT\_PRIO-1 MAX\_RT\_PRIO的定义参见 include/linux/sched.h

1611 #define MAX\_USER\_RT\_PRIO 100

1612 #define MAX\_RT\_PRIO MAX\_USER\_RT\_PRIO

nice值在内核中的范围是 MAX\_RT\_PRIO～MAX\_RT\_PRIO+40 即 MAX\_RT\_PRIO～MAX\_PRIO

1614 #define MAX\_PRIO (MAX\_RT\_PRIO + 40)

第二个问题，一个进程同时有2种优先级怎么办？

答案很简单，就是一个进程不可能有2个优先级。一个进程有了实时优先级就没有Nice值，有了Nice值就没有实时优先级。

我们可以通过以下命令查看进程的实时优先级和Nice值：(其中RTPRIO是实时优先级，NI是Nice值)

[复制代码](javascript:void(0);)

$ ps -eo state,uid,pid,ppid,rtprio,ni,time,comm

S UID PID PPID RTPRIO NI TIME COMMAND

S 0 1 0 - 0 00:00:00 systemd

S 0 2 0 - 0 00:00:00 kthreadd

S 0 3 2 - 0 00:00:00 ksoftirqd/0

S 0 6 2 99 - 00:00:00 migration/0

S 0 7 2 99 - 00:00:00 watchdog/0

S 0 8 2 99 - 00:00:00 migration/1

S 0 10 2 - 0 00:00:00 ksoftirqd/1

S 0 12 2 99 - 00:00:00 watchdog/1

S 0 13 2 99 - 00:00:00 migration/2

S 0 15 2 - 0 00:00:00 ksoftirqd/2

S 0 16 2 99 - 00:00:00 watchdog/2

S 0 17 2 99 - 00:00:00 migration/3

S 0 19 2 - 0 00:00:00 ksoftirqd/3

S 0 20 2 99 - 00:00:00 watchdog/3

S 0 21 2 - -20 00:00:00 cpuset

S 0 22 2 - -20 00:00:00 khelper

[复制代码](javascript:void(0);)

#### 2.2 关于时间片

有了优先级，可以决定谁先运行了。但是对于调度程序来说，并不是运行一次就结束了，还必须知道间隔多久进行下次调度。

于是就有了时间片的概念。时间片是一个数值，表示一个进程被抢占前能持续运行的时间。

也可以认为是进程在下次调度发生前运行的时间(除非进程主动放弃CPU，或者有实时进程来抢占CPU)。

时间片的大小设置并不简单，设大了，系统响应变慢(调度周期长)；设小了，进程频繁切换带来的处理器消耗。默认的时间片一般是10ms

#### 2.3 调度实现原理（基于优先级和时间片）

下面举个直观的例子来说明：

假设系统中只有3个进程ProcessA(NI=+10)，ProcessB(NI=0)，ProcessC(NI=-10)，NI表示进程的nice值，时间片=10ms

1) 调度前，把进程优先级按一定的权重映射成时间片(这里假设优先级高一级相当于多5msCPU时间)。

    假设ProcessA分配了一个时间片10ms，那么ProcessB的优先级比ProcessA高10(nice值越小优先级越高)，ProcessB应该分配10\*5+10=60ms，以此类推，ProcessC分配20\*5+10=110ms

2) 开始调度时，优先调度分配CPU时间多的进程。由于ProcessA(10ms),ProcessB(60ms),ProcessC(110ms)。显然先调度ProcessC

3) 10ms(一个时间片)后，再次调度时，ProcessA(10ms),ProcessB(60ms),ProcessC(100ms)。ProcessC刚运行了10ms，所以变成100ms。此时仍然先调度ProcessC

4) 再调度4次后(4个时间片)，ProcessA(10ms),ProcessB(60ms),ProcessC(60ms)。此时ProcessB和ProcessC的CPU时间一样，这时得看ProcessB和ProcessC谁在CPU运行队列的前面，假设ProcessB在前面，则调度ProcessB

5) 10ms(一个时间片)后，ProcessA(10ms),ProcessB(50ms),ProcessC(60ms)。再次调度ProcessC

6) ProcessB和ProcessC交替运行，直至ProcessA(10ms),ProcessB(10ms),ProcessC(10ms)。

    这时得看ProcessA，ProcessB，ProcessC谁在CPU运行队列的前面就先调度谁。这里假设调度ProcessA

7) 10ms(一个时间片)后，ProcessA(时间片用完后退出),ProcessB(10ms),ProcessC(10ms)。

8) 再过2个时间片，ProcessB和ProcessC也运行完退出。

这个例子很简单，主要是为了说明调度的原理，实际的调度算法虽然不会这么简单，但是基本的实现原理也是类似的：

1）确定每个进程能占用多少CPU时间(这里确定CPU时间的算法有很多，根据不同的需求会不一样)

2）占用CPU时间多的先运行

3）运行完后，扣除运行进程的CPU时间，再回到 1）

### ****3. Linux上调度实现的方法****

Linux上的调度算法是不断发展的，在2.6.23内核以后，采用了“完全公平调度算法”，简称CFS。

CFS算法在分配每个进程的CPU时间时，不是分配给它们一个绝对的CPU时间，而是根据进程的优先级分配给它们一个占用CPU时间的百分比。

比如ProcessA(NI=1)，ProcessB(NI=3)，ProcessC(NI=6)，在CFS算法中，分别占用CPU的百分比为：ProcessA(10%)，ProcessB(30%)，ProcessC(60%)

因为总共是100%，ProcessB的优先级是ProcessA的3倍，ProcessC的优先级是ProcessA的6倍。

Linux上的CFS算法主要有以下步骤：(还是以ProcessA(10%)，ProcessB(30%)，ProcessC(60%)为例)

1)计算每个进程的vruntime(注1)，通过update\_curr()函数更新进程的vruntime。

2)选择具有最小vruntime的进程投入运行。（注2）

3)进程运行完后，更新进程的vruntime，转入步骤2) （注3）

**注1.** 这里的vruntime是进程虚拟运行的时间的总和。vruntime定义在：kernel/sched\_fair.c 文件的 struct sched\_entity 中。

**注2.** 这里有点不好理解，根据vruntime来选择要运行的进程，似乎和每个进程所占的CPU时间百分比没有关系了。

1）比如先运行ProcessC，(vr是vruntime的缩写)，则10ms后：ProcessA(vr=0)，ProcessB(vr=0)，ProcessC(vr=10)

2）那么下次调度只能运行ProcessA或者ProcessB。(因为会选择具有最小vruntime的进程)

长时间来看的话，ProcessA、ProcessB、ProcessC是公平的交替运行的，和优先级没有关系。

而实际上**vruntime**并不是实际的运行时间，它是**实际运行时间进行加权运算**后的结果。

比如上面3个进程中ProcessA(10%)只分配了CPU总的处理时间的10%，那么ProcessA运行10ms的话，它的vruntime会增加100ms。

以此类推，ProcessB运行10ms的话，它的vruntime会增加(100/3)ms,ProcessC运行10ms的话，它的vruntime会增加(100/6)ms。

实际的运行时，由于ProcessC的vruntime增加的最慢，所以它会获得最多的CPU处理时间。

上面的加权算法是我自己为了理解方便简化的，Linux对vruntime的加权方法还得去看源码^-^

**注3.**Linux为了能快速的找到具有最小vruntime，将所有的进程的存储在一个红黑树中。这样树的最左边的叶子节点就是具有最小vruntime的进程，新的进程加入或有旧的进程退出时都会更新这棵树。

其实Linux上的调度器是以模块方式提供的，每个调度器有不同的优先级，所以可以同时存在多种调度算法。

每个进程可以选择自己的调度器，Linux调度时，首先按调度器的优先级选择一个调度器，再选择这个调度器下的进程。

### ****4. 调度相关的系统调用****

调度相关的系统调用主要有2类：

1) 与调度策略和进程优先级相关 (就是上面的提到的各种参数，优先级，时间片等等) - 下表中的前8个

2) 与处理器相关 - 下表中的最后3个

|  |  |
| --- | --- |
| **系统调用** | **描述** |
| nice() | 设置进程的nice值 |
| sched\_setscheduler() | 设置进程的调度策略，即设置进程采取何种调度算法 |
| sched\_getscheduler() | 获取进程的调度算法 |
| sched\_setparam() | 设置进程的实时优先级 |
| sched\_getparam() | 获取进程的实时优先级 |
| sched\_get\_priority\_max() | 获取实时优先级的最大值，由于用户权限的问题，非root用户并不能设置实时优先级为99 |
| sched\_get\_priority\_min() | 获取实时优先级的最小值，理由与上面类似 |
| sched\_rr\_get\_interval() | 获取进程的时间片 |
| sched\_setaffinity() | 设置进程的处理亲和力，其实就是保存在task\_struct中的cpu\_allowed这个掩码标志。该掩码的每一位对应一个系统中可用的处理器，默认所有位都被设置，即该进程可以再系统中所有处理器上执行。  用户可以通过此函数设置不同的掩码，使得进程只能在系统中某一个或某几个处理器上运行。 |
| sched\_getaffinity() | 获取进程的处理亲和力 |
| sched\_yield() | 暂时让出处理器 |

# [系统调用](http://www.cnblogs.com/wang_yb/archive/2012/09/17/2688263.html)

**主要内容：**

1. 什么是系统调用
2. Linux上的系统调用实现原理
3. 一个简单的系统调用的实现

### ****1. 什么是系统调用****

简单来说，系统调用就是用户程序和硬件设备之间的桥梁。

用户程序在需要的时候，通过系统调用来使用硬件设备。

系统调用的存在，有以下重要的意义:

1）用户程序通过系统调用来使用硬件，而不用关心具体的硬件设备，这样大大简化了用户程序的开发。

    比如：用户程序通过write()系统调用就可以将数据写入文件，而不必关心文件是在磁盘上还是软盘上，或者其他存储上。

2）系统调用使得用户程序有更好的可移植性。

    只要操作系统提供的系统调用接口相同，用户程序就可在不用修改的情况下，从一个系统迁移到另一个操作系统。

3）系统调用使得内核能更好的管理用户程序，增强了系统的稳定性。

    因为系统调用是内核实现的，内核通过系统调用来控制开放什么功能及什么权限给用户程序。

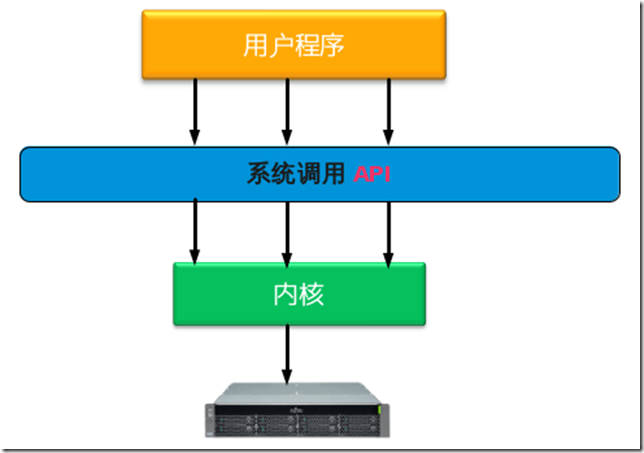
    这样可以避免用户程序不正确的使用硬件设备，从而破坏了其他程序。

4）系统调用有效的分离了用户程序和内核的开发。

    用户程序只需关心系统调用API，通过这些API来开发自己的应用，不用关心API的具体实现。

    内核则只要关心系统调用API的实现，而不必管它们是被如何调用的。

用户程序，系统调用，内核，硬件设备的调用关系如下图：

[](http://images.cnblogs.com/cnblogs_com/wang_yb/201209/201209170957496310.png)

### ****2. Linux上的系统调用实现原理****

要想实现系统调用，主要实现以下几个方面：

1. 通知内核调用一个哪个系统调用
2. 用户程序把系统调用的参数传递给内核
3. 用户程序获取内核返回的系统调用返回值

下面看看Linux是如何实现上面3个功能的。

#### 2.1 通知内核调用一个哪个系统调用

每个系统调用都有一个系统调用号，系统调用发生时，内核就是根据传入的系统调用号来知道是哪个系统调用的。

在x86架构中，用户空间将系统调用号是放在eax中的，系统调用处理程序通过eax取得系统调用号。

系统调用号定义在内核代码：arch/alpha/include/asm/unistd.h 中，可以看出linux的系统调用不是很多。

#### 2.2 用户程序把系统调用的参数传递给内核

系统调用的参数也是通过寄存器传给内核的，在x86系统上，系统调用的前5个参数放在ebx,ecx,edx,esi和edi中，如果参数多的话，还需要用个单独的寄存器存放指向所有参数在用户空间地址的指针。

一般的系统调用都是通过C库(最常用的是glibc库)来访问的，Linux内核提供一个从用户程序直接访问系统调用的方法。

参见内核代码：arch/cris/include/arch-v10/arch/unistd.h

里面定义了6个宏，分别可以调用参数个数为0～6的系统调用

[复制代码](javascript:void(0);)

\_syscall0(type,name)

\_syscall1(type,name,type1,arg1)

\_syscall2(type,name,type1,arg1,type2,arg2)

\_syscall3(type,name,type1,arg1,type2,arg2,type3,arg3)

\_syscall4(type,name,type1,arg1,type2,arg2,type3,arg3,type4,arg4)

\_syscall5(type,name,type1,arg1,type2,arg2,type3,arg3,type4,arg4,type5,arg5)

\_syscall6(type,name,type1,arg1,type2,arg2,type3,arg3,type4,arg4,type5,arg5,type6,arg6)

[复制代码](javascript:void(0);)

超过6个参数的系统调用很罕见，所以这里只定义了6个。

#### 2.3 用户程序获取内核返回的系统调用返回值

获取系统调用的返回值也是通过寄存器，在x86系统上，返回值放在eax中。

### ****3. 一个简单的系统调用的实现****

了解了Linux上系统调用的原理，下面就可以自己来实现一个简单的系统调用。

#### 3.1 环境准备

为了不破坏现有系统，我是用虚拟机来实验的。

主机：fedora16 x86\_64系统 + kvm(一种虚拟技术，就像virtualbox，vmware等)

虚拟机: 也是安装fedora16 x86\_64系统(通过virt-manager很容易安装一个系统)

下载内核源码：www.kernel.org  下载最新的就行

#### 3.2 修改内核源码中的相应文件

主要修改以下文件：

[复制代码](javascript:void(0);)

arch/x86/ia32/ia32entry.S

arch/x86/include/asm/unistd\_32.h

arch/x86/include/asm/unistd\_64.h

arch/x86/kernel/syscall\_table\_32.S

include/asm-generic/unistd.h

include/linux/syscalls.h

kernel/sys.c

[复制代码](javascript:void(0);)

我在sys.c中追加了2个函数:sys\_foo和sys\_bar

如果是在x86\_64的内核中增加一个系统调用，只需修改 arch/x86/include/asm/unistd\_64.h，比如sys\_bar。

修改内容参见下面的diff文件：

[复制代码](javascript:void(0);)

diff -r new/arch/x86/ia32/ia32entry.S old/arch/x86/ia32/ia32entry.S

855d854

< .quad sys\_foo

diff -r new/arch/x86/include/asm/unistd\_32.h old/arch/x86/include/asm/unistd\_32.h

357d356

< #define \_\_NR\_foo 349

361c360

< #define NR\_syscalls 350

---

> #define NR\_syscalls 349

diff -r new/arch/x86/include/asm/unistd\_64.h old/arch/x86/include/asm/unistd\_64.h

689,692d688

< #define \_\_NR\_foo 312

< \_\_SYSCALL(\_\_NR\_foo, sys\_foo)

< #define \_\_NR\_bar 313

< \_\_SYSCALL(\_\_NR\_bar, sys\_bar)

diff -r new/arch/x86/kernel/syscall\_table\_32.S old/arch/x86/kernel/syscall\_table\_32.S

351d350

< .long sys\_foo

diff -r new/include/asm-generic/unistd.h old/include/asm-generic/unistd.h

694,695d693

< #define \_\_NR\_foo 272

< \_\_SYSCALL(\_\_NR\_foo, sys\_foo)

698c696

< #define \_\_NR\_syscalls 273

---

> #define \_\_NR\_syscalls 272

diff -r new/kernel/sys.c old/kernel/sys.c

1920,1928d1919

<

< asmlinkage long sys\_foo(void)

< {

< return 1112223334444555;

< }

< asmlinkage long sys\_bar(void)

< {

< return 1234567890;

< }

[复制代码](javascript:void(0);)

#### 3.3 编译内核

#cd linux-3.2.28

#make menuconfig (选择要编译参数，如果不熟悉内核编译，用默认选项即可)

#make all (这一步真的时间很长......)

#make modules\_install

#make install (这一步会把新的内核加到启动项中)

#reboot (重启系统进入新的内核)

#### 3.4 编写调用的系统调用的代码

[复制代码](javascript:void(0);)

#include <unistd.h>

#include <sys/syscall.h>

#include <string.h>

#include <stdio.h>

#include <errno.h>

#define \_\_NR\_foo 312

#define \_\_NR\_bar 313

int main()

{

printf ("result foo is %ld\n", syscall(\_\_NR\_foo));

printf("%s\n", strerror(errno));

printf ("result bar is %ld\n", syscall(\_\_NR\_bar));

printf("%s\n", strerror(errno));

return 0;

}

[复制代码](javascript:void(0);)

编译运行上面的代码：

#gcc test.c -o test

#./test

运行结果如下：

result foo is 1112223334444555

Success

result bar is 1234567890

Success

# [内核数据结构](http://www.cnblogs.com/wang_yb/archive/2013/04/16/3023892.html)

内核数据结构贯穿于整个内核代码中，这里介绍4个基本的内核数据结构。

利用这4个基本的数据结构，可以在编写内核代码时节约大量时间。

**主要内容：**

* 链表
* 队列
* 映射
* 红黑树

### 1. 链表

链表是linux内核中最简单，同时也是应用最广泛的数据结构。

内核中定义的是双向链表。

#### 1.1 头文件简介

内核中关于链表定义的代码位于： include/linux/list.h

list.h文件中对每个函数都有注释，这里就不详细说了。

其实刚开始只要先了解一个常用的链表操作（追加，删除，遍历）的实现方法，

其他方法基本都是基于这些常用操作的。

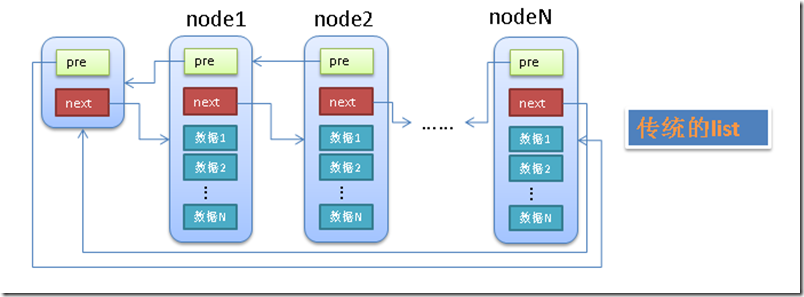
#### 1.2 链表代码的注意点

在阅读list.h文件之前，有一点必须注意：linux内核中的链表使用方法和一般数据结构中定义的链表是有所不同的。

一般的双向链表一般是如下的结构，

* 有个单独的头结点(head)
* 每个节点(node)除了包含必要的数据之外，还有2个指针(pre,next)
* pre指针指向前一个节点(node)，next指针指向后一个节点(node)
* 头结点(head)的pre指针指向链表的最后一个节点
* 最后一个节点的next指针指向头结点(head)

具体见下图：

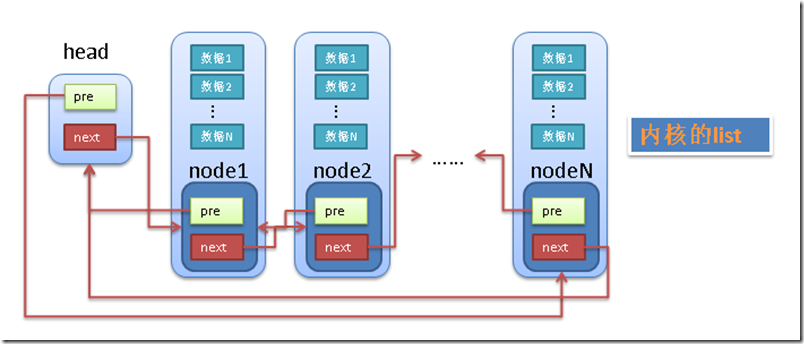
[](https://images0.cnblogs.com/blog/83005/201304/16132037-c83d5c19780d4c13ade4964ed38e6af7.png)

传统的链表有个最大的缺点就是不好共通化，因为每个node中的data1，data2等等都是不确定的(无论是个数还是类型)。

linux中的链表巧妙的解决了这个问题，linux的链表不是将用户数据保存在链表节点中，而是将链表节点保存在用户数据中。

linux的链表节点只有2个指针(pre和next)，这样的话，链表的节点将独立于用户数据之外，便于实现链表的共同操作。

具体见下图：

[](https://images0.cnblogs.com/blog/83005/201304/16132043-3ae739a1effd40a4afe22a5661018919.png)

linux链表中的最大问题是怎样通过链表的节点来取得用户数据？

和传统的链表不同，linux的链表节点(node)中没有包含用户的用户data1，data2等。

整个list.h文件中，我觉得最复杂的代码就是获取用户数据的宏定义

#define list\_entry(ptr, type, member) \

container\_of(ptr, type, member)

这个宏没什么特别的，主要是container\_of这个宏

#define container\_of(ptr, type, member) ({ \

const typeof(((type \*)0)->member)\*\_\_mptr = (ptr); \

(type \*)((char \*)\_\_mptr - offsetof(type, member)); })

这里面的type一般是个结构体，也就是包含用户数据和链表节点的结构体。

ptr是指向type中链表节点的指针

member则是type中定义链表节点是用的名字

比如：

struct student

{

int id;

char\* name;

struct list\_head list;

};

* type是struct student
* ptr是指向stuct list的指针，也就是指向member类型的指针
* member就是 list

下面分析一下container\_of宏:

[复制代码](javascript:void(0);)

// 步骤1：将数字0强制转型为type\*，然后取得其中的member元素

((type \*)0)->member // 相当于((struct student \*)0)->list

// 步骤2：定义一个临时变量\_\_mptr，并将其也指向ptr所指向的链表节点

const typeof(((type \*)0)->member)\*\_\_mptr = (ptr);

// 步骤3：计算member字段距离type中第一个字段的距离，也就是type地址和member地址之间的差

// offset(type, member)也是一个宏，定义如下：

#define offsetof(TYPE, MEMBER) ((size\_t) &((TYPE \*)0)->MEMBER)

// 步骤4：将\_\_mptr的地址 - type地址和member地址之间的差

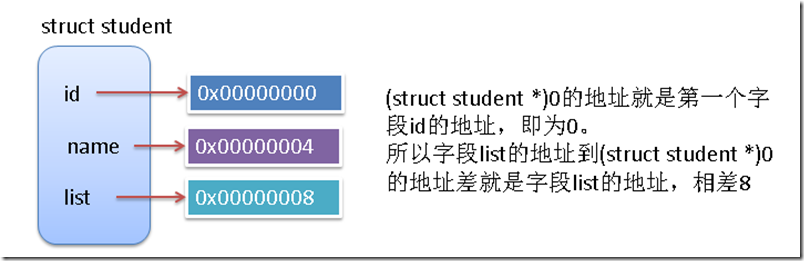
// 其实也就是获取type的地址

[复制代码](javascript:void(0);)

步骤1，2，4比较容易理解，下面的图以sturct student为例进行说明步骤3：

首先需要知道 ((TYPE \*)0) 表示将地址0转换为 TYPE 类型的地址

由于TYPE的地址是0，所以((TYPE \*)0)->MEMBER 也就是 MEMBER的地址和TYPE地址的差，如下图所示：

[](https://images0.cnblogs.com/blog/83005/201304/16132050-2b81fd5bac9040c28c3cea1d9a9e3365.png)

#### 1.3 使用示例

构造了一个内核模块来实际使用一下内核中的链表，代码在CentOS6.3 x64上运行通过。

C代码：

[复制代码](javascript:void(0);)

#include<linux/init.h>

#include<linux/slab.h>

#include<linux/module.h>

#include<linux/kernel.h>

#include<linux/list.h>

MODULE\_LICENSE("Dual BSD/GPL");

struct student

{

int id;

char\* name;

struct list\_head list;

};

void print\_student(struct student\*);

static int testlist\_init(void)

{

struct student \*stu1, \*stu2, \*stu3, \*stu4;

struct student \*stu;

// init a list head

LIST\_HEAD(stu\_head);

// init four list nodes

stu1 = kmalloc(sizeof(\*stu1), GFP\_KERNEL);

stu1->id = 1;

stu1->name = "wyb";

INIT\_LIST\_HEAD(&stu1->list);

stu2 = kmalloc(sizeof(\*stu2), GFP\_KERNEL);

stu2->id = 2;

stu2->name = "wyb2";

INIT\_LIST\_HEAD(&stu2->list);

stu3 = kmalloc(sizeof(\*stu3), GFP\_KERNEL);

stu3->id = 3;

stu3->name = "wyb3";

INIT\_LIST\_HEAD(&stu3->list);

stu4 = kmalloc(sizeof(\*stu4), GFP\_KERNEL);

stu4->id = 4;

stu4->name = "wyb4";

INIT\_LIST\_HEAD(&stu4->list);

// add the four nodes to head

list\_add (&stu1->list, &stu\_head);

list\_add (&stu2->list, &stu\_head);

list\_add (&stu3->list, &stu\_head);

list\_add (&stu4->list, &stu\_head);

// print each student from 4 to 1

list\_for\_each\_entry(stu, &stu\_head, list)

{

print\_student(stu);

}

// print each student from 1 to 4

list\_for\_each\_entry\_reverse(stu, &stu\_head, list)

{

print\_student(stu);

}

// delete a entry stu2

list\_del(&stu2->list);

list\_for\_each\_entry(stu, &stu\_head, list)

{

print\_student(stu);

}

// replace stu3 with stu2

list\_replace(&stu3->list, &stu2->list);

list\_for\_each\_entry(stu, &stu\_head, list)

{

print\_student(stu);

}

return 0;

}

static void testlist\_exit(void)

{

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

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

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

}

void print\_student(struct student \*stu)

{

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

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

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

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

}

module\_init(testlist\_init);

module\_exit(testlist\_exit);

[复制代码](javascript:void(0);)

Makefile：

[复制代码](javascript:void(0);)

obj-m += testlist.o

#generate the path

CURRENT\_PATH:=$(shell pwd)

#the current kernel version number

LINUX\_KERNEL:=$(shell uname -r)

#the absolute path

LINUX\_KERNEL\_PATH:=/usr/src/kernels/$(LINUX\_KERNEL)

#complie object

all:

make -C $(LINUX\_KERNEL\_PATH) M=$(CURRENT\_PATH) modules

rm -rf modules.order Module.symvers .\*.cmd \*.o \*.mod.c .tmp\_versions \*.unsigned

#clean

clean:

rm -rf modules.order Module.symvers .\*.cmd \*.o \*.mod.c \*.ko .tmp\_versions \*.unsigned

[复制代码](javascript:void(0);)

安装,卸载内核模块以及查看内核模块的运行结果：

insmod testlist.ko

rmmod testlist

dmesg | tail -100

### 2. 队列

内核中的队列是以字节形式保存数据的，所以获取数据的时候，需要知道数据的大小。

如果从队列中取得数据时指定的大小不对的话，取得数据会不完整或过大。

#### 2.1 头文件简介

内核中关于队列定义的头文件位于：<linux/kfifo.h> include/linux/kfifo.h

头文件中定义的函数的实现位于：kernel/kfifo.c

#### 2.2 队列代码的注意点

内核队列编程需要注意的是：

* 队列的size在初始化时，始终设定为2的n次方
* 使用队列之前将队列结构体中的锁(spinlock)释放

#### 2.3 使用示例

构造了一个内核模块来实际使用一下内核中的队列，代码在CentOS6.3 x64上运行通过。

C代码:

[复制代码](javascript:void(0);)

#include "kn\_common.h"

MODULE\_LICENSE("Dual BSD/GPL");

struct student

{

int id;

char\* name;

};

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

static int testkfifo\_init(void)

{

struct kfifo \*fifo;

struct student \*stu1, \*stu2, \*stu3, \*stu4;

struct student \*stu\_tmp;

char\* c\_tmp;

int i;

// !!importent init a unlocked lock

spinlock\_t sl = SPIN\_LOCK\_UNLOCKED;

// init kfifo

fifo = kfifo\_alloc(4\*sizeof(struct student), GFP\_KERNEL, &sl);

stu1 = kmalloc(sizeof(struct student), GFP\_KERNEL);

stu1->id = 1;

stu1->name = "wyb1";

kfifo\_put(fifo, (char \*)stu1, sizeof(struct student));

stu2 = kmalloc(sizeof(struct student), GFP\_KERNEL);

stu2->id = 1;

stu2->name = "wyb2";

kfifo\_put(fifo, (char \*)stu2, sizeof(struct student));

stu3 = kmalloc(sizeof(struct student), GFP\_KERNEL);

stu3->id = 1;

stu3->name = "wyb3";

kfifo\_put(fifo, (char \*)stu3, sizeof(struct student));

stu4 = kmalloc(sizeof(struct student), GFP\_KERNEL);

stu4->id = 1;

stu4->name = "wyb4";

kfifo\_put(fifo, (char \*)stu4, sizeof(struct student));

c\_tmp = kmalloc(sizeof(struct student), GFP\_KERNEL);

printk(KERN\_ALERT "current fifo length is : %d\n", kfifo\_len(fifo));

for (i=0; i < 4; i++) {

kfifo\_get(fifo, c\_tmp, sizeof(struct student));

stu\_tmp = (struct student \*)c\_tmp;

print\_student(stu\_tmp);

printk(KERN\_ALERT "current fifo length is : %d\n", kfifo\_len(fifo));

}

printk(KERN\_ALERT "current fifo length is : %d\n", kfifo\_len(fifo));

kfifo\_free(fifo);

kfree(c\_tmp);

return 0;

}

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

{

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

print\_current\_time(1);

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

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

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

}

static void testkfifo\_exit(void)

{

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

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

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

}

module\_init(testkfifo\_init);

module\_exit(testkfifo\_exit);

[复制代码](javascript:void(0);)

其中引用的kn\_common.h文件：

[复制代码](javascript:void(0);)

#include<linux/init.h>

#include<linux/slab.h>

#include<linux/module.h>

#include<linux/kernel.h>

#include<linux/kfifo.h>

#include<linux/time.h>

void print\_current\_time(int);

[复制代码](javascript:void(0);)

kn\_common.h对应的kn\_common.c：

[复制代码](javascript:void(0);)

#include "kn\_common.h"

void print\_current\_time(int is\_new\_line)

{

struct timeval \*tv;

struct tm \*t;

tv = kmalloc(sizeof(struct timeval), GFP\_KERNEL);

t = kmalloc(sizeof(struct tm), GFP\_KERNEL);

do\_gettimeofday(tv);

time\_to\_tm(tv->tv\_sec, 0, t);

printk(KERN\_ALERT "%ld-%d-%d %d:%d:%d",

t->tm\_year + 1900,

t->tm\_mon + 1,

t->tm\_mday,

(t->tm\_hour + 8) % 24,

t->tm\_min,

t->tm\_sec);

if (is\_new\_line == 1)

printk(KERN\_ALERT "\n");

kfree(tv);

kfree(t);

}

[复制代码](javascript:void(0);)

Makefile：

[复制代码](javascript:void(0);)

obj-m += fifo.o

fifo-objs := testkfifo.o kn\_common.o

#generate the path

CURRENT\_PATH:=$(shell pwd)

#the current kernel version number

LINUX\_KERNEL:=$(shell uname -r)

#the absolute path

LINUX\_KERNEL\_PATH:=/usr/src/kernels/$(LINUX\_KERNEL)

#complie object

all:

make -C $(LINUX\_KERNEL\_PATH) M=$(CURRENT\_PATH) modules

rm -rf modules.order Module.symvers .\*.cmd \*.o \*.mod.c .tmp\_versions \*.unsigned

#clean

clean:

rm -rf modules.order Module.symvers .\*.cmd \*.o \*.mod.c \*.ko .tmp\_versions \*.unsigned

[复制代码](javascript:void(0);)

安装,卸载内核模块以及查看内核模块的运行结果：

insmod fifo.ko

rmmod fifo

dmesg | tail -40

### 3. 映射

映射的有点想其他语言(C#或者python)中的字典类型，每个唯一的id对应一个自定义的数据结构。

#### 3.1 头文件简介

内核中关于映射定义的头文件位于：<linux/idr.h> include/linux/idr.h

头文件中定义的函数的实现位于：lib/idr.c

#### 3.2 映射代码的注意点

映射的使用需要注意的是，给自定义的数据结构申请一个id的时候，不能直接申请id，先要分配id(函数idr\_pre\_get)，分配成功后，在获取一个id(函数idr\_get\_new)。

idr的结构比较复杂，我也没有很好的理解，但是csdn上有篇介绍linux idr结构的博客写的挺好，图文并茂：<http://blog.csdn.net/paomadi/article/details/8539794>

#### 3.3 使用示例

构造了一个内核模块来实际使用一下内核中的映射，代码在CentOS6.3 x64上运行通过。

C代码:

[复制代码](javascript:void(0);)

#include<linux/idr.h>

#include "kn\_common.h"

MODULE\_LICENSE("Dual BSD/GPL");

struct student

{

int id;

char\* name;

};

static int print\_student(int, void\*, void\*);

static int testidr\_init(void)

{

DEFINE\_IDR(idp);

struct student \*stu[4];

// struct student \*stu\_tmp;

int id, ret, i;

// init 4 struct student

for (i=0; i<4; i++) {

stu[i] = kmalloc(sizeof(struct student), GFP\_KERNEL);

stu[i]->id = i;

stu[i]->name = "wyb";

}

// add 4 student to idr

print\_current\_time(0);

for (i=0; i < 4; i++) {

do {

if (!idr\_pre\_get(&idp, GFP\_KERNEL))

return -ENOSPC;

ret = idr\_get\_new(&idp, stu[i], &id);

printk(KERN\_ALERT "id=%d\n", id);

} while(ret == -EAGAIN);

}

// display all student in idr

idr\_for\_each(&idp, print\_student, NULL);

idr\_destroy(&idp);

kfree(stu[0]);

kfree(stu[1]);

kfree(stu[2]);

kfree(stu[3]);

return 0;

}

static int print\_student(int id, void \*p, void \*data)

{

struct student\* stu = p;

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

print\_current\_time(0);

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

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

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

return 0;

}

static void testidr\_exit(void)

{

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

print\_current\_time(0);

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

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

}

module\_init(testidr\_init);

module\_exit(testidr\_exit);

[复制代码](javascript:void(0);)

***注：***其中用到的kn\_common.h和kn\_common.c文件与队列的示例中一样。

Makefile：

[复制代码](javascript:void(0);)

obj-m += idr.o

idr-objs := testidr.o kn\_common.o

#generate the path

CURRENT\_PATH:=$(shell pwd)

#the current kernel version number

LINUX\_KERNEL:=$(shell uname -r)

#the absolute path

LINUX\_KERNEL\_PATH:=/usr/src/kernels/$(LINUX\_KERNEL)

#complie object

all:

make -C $(LINUX\_KERNEL\_PATH) M=$(CURRENT\_PATH) modules

rm -rf modules.order Module.symvers .\*.cmd \*.o \*.mod.c .tmp\_versions \*.unsigned

#clean

clean:

rm -rf modules.order Module.symvers .\*.cmd \*.o \*.mod.c \*.ko .tmp\_versions \*.unsigned

[复制代码](javascript:void(0);)

安装,卸载内核模块以及查看内核模块的运行结果：

insmod idr.ko

rmmod idr

dmesg | tail -30

### 4. 红黑树

红黑树由于节点颜色的特性，保证其是一种自平衡的二叉搜索树。

红黑树的一系列规则虽然实现起来比较复杂，但是遵循起来却比较简单，而且红黑树的插入，删除性能也还不错。

所以红黑树在内核中的应用非常广泛，掌握好红黑树，即有利于阅读内核源码，也可以在自己的代码中借鉴这种数据结构。

红黑树必须满足的规则：

* 所有节点都有颜色，要么红色，要么黑色
* 根节点是黑色，所有叶子节点也是黑色
* 叶子节点中不包含数据
* 非叶子节点都有2个子节点
* 如果一个节点是红色，那么它的父节点和子节点都是黑色的
* 从任何一个节点开始，到其下叶子节点的路径中都包含相同数目的黑节点

红黑树中最长的路径就是红黑交替的路径，最短的路径是全黑节点的路径，再加上根节点和叶子节点都是黑色，

从而可以保证红黑树中最长路径的长度不会超过最短路径的2倍。

#### 4.1 头文件简介

内核中关于红黑树定义的头文件位于：<linux/rbtree.h> include/linux/rbtree.h

头文件中定义的函数的实现位于：lib/rbtree.c

#### 4.2 红黑树代码的注意点

内核中红黑树的使用和链表(list)有些类似，是将红黑树的节点放入自定义的数据结构中来使用的。

首先需要注意的一点是红黑树节点的定义：

[复制代码](javascript:void(0);)

struct rb\_node

{

unsigned long rb\_parent\_color;

#define RB\_RED 0

#define RB\_BLACK 1

struct rb\_node \*rb\_right;

struct rb\_node \*rb\_left;

} \_\_attribute\_\_((aligned(sizeof(long))));

[复制代码](javascript:void(0);)

刚开始看到这个定义的时候，我觉得很奇怪，等到看懂了之后，才知道原来作者巧妙的利用内存对齐来将2个内容存入到一个字段中（不服不行啊^\_^!）。

字段 rb\_parent\_color 中保存了2个信息：

1. 父节点的地址
2. 本节点的颜色

这2个信息是如何存入一个字段的呢？主要在于 \_\_attribute\_\_((aligned(sizeof(long))));

这行代码的意思就是 struct rb\_node 在内存中的地址需要按照4 bytes或者8 bytes对齐。

***注：***sizeof(long) 在32bit系统中是4 bytes，在64bit系统中是8 bytes

struct rb\_node的地址按4 bytes对齐，意味着分配的地址都是4的倍数。

4 的二进制为 100 ，所以申请分配的 struct rb\_node 的地址的最后2位始终是零，

struct rb\_node 的字段 rb\_parent\_color 就是利用最后一位来保存节点的颜色信息的。

明白了这点之后，rb\_tree.h 中很多宏的定义也就很好懂了。

[复制代码](javascript:void(0);)

/\* rb\_parent\_color 保存了父节点的地址和本节点的颜色 \*/

/\* 将 rb\_parent\_color 的最后2位置成0，即将颜色信息去掉，剩下的就是parent节点的地址 \*/

#define rb\_parent(r) ((struct rb\_node \*)((r)->rb\_parent\_color & ~3))

/\* 取得 rb\_parent\_color 二进制表示的最后一位，即用于保存颜色信息的那一位 \*/

#define rb\_color(r) ((r)->rb\_parent\_color & 1)

/\* 将 rb\_parent\_color 二进制表示的最后一位置为0，即置为红色 \*/

#define rb\_set\_red(r) do { (r)->rb\_parent\_color &= ~1; } while (0)

/\* 将 rb\_parent\_color 二进制表示的最后一位置为1，即置为黑色 \*/

#define rb\_set\_black(r) do { (r)->rb\_parent\_color |= 1; } while (0)

[复制代码](javascript:void(0);)

还有需要重点看的就是rb\_tree.c中的5个函数，下面对这5个函数进行一些注释：

**函数1**：左旋操作，当右子树的长度过大导致树不平衡时，进行左旋操作

[复制代码](javascript:void(0);)

/\*

\* 左旋操作其实就3个动作：见图left

\* 1. node的右子树关联到right的左子树

\* 2. right的左子树关联到node

\* 3. right取代node的位置

\* 其他带代码都是一些相应的parent指针的变化

\*/

static void \_\_rb\_rotate\_left(struct rb\_node \*node, struct rb\_root \*root)

{

/\* 初始化相对于node节点的父节点(图中的P)和右节点(图中的R) \*/

struct rb\_node \*right = node->rb\_right;

struct rb\_node \*parent = rb\_parent(node);

/\* 步骤1 \*/

if ((node->rb\_right = right->rb\_left))

rb\_set\_parent(right->rb\_left, node);

/\* 步骤2 \*/

right->rb\_left = node;

rb\_set\_parent(right, parent);

/\* node的parent NOT NULL 时，right取代原先的node的位置 \*/

if (parent)

{

if (node == parent->rb\_left)

parent->rb\_left = right;

else

parent->rb\_right = right;

}

/\* node的parent NULL 时，说明node原先时root节点，将新的root指向root即可 \*/

else

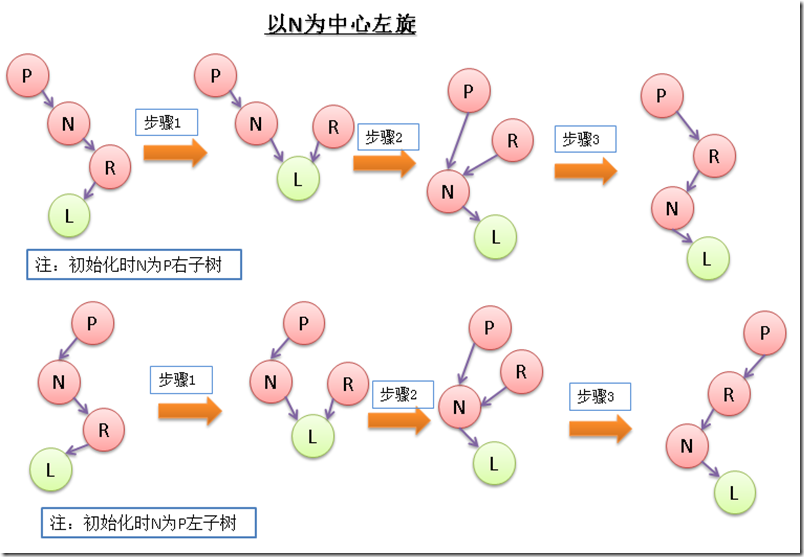
root->rb\_node = right;

rb\_set\_parent(node, right);

}

[复制代码](javascript:void(0);)

左旋操作图解:

[](https://images0.cnblogs.com/blog/83005/201304/16132101-5ea192c2655d45ecb6093984ede5e3a6.png)

**函数2**：右旋操作，和左旋操作类似。

**函数3**：追加节点后，设置此节点的颜色。

[复制代码](javascript:void(0);)

/\*

\* 本函数没有插入节点的功能，只是在插入新节点后，设置新节点的颜色，从而保证红黑树的平衡性。

\* 新插入的节点默认都是红色的。

\*

\* 下面的代码看着复杂，其实只要时时记住红黑树的几个重要特性，就会发现下面的都是在尽量保持住红黑树的这些特性。

\* 1. 无论从哪个节点开始，到其叶子节点的路径中包含的黑色节点个数时一样的

\* 2. 不能有连续的2个红色节点，即父节点和子节点不能同时为红色

\* 所以最简单的情况就是：插入节点的父节点是黑色的。那么插入一个红节点后不会有任何影响。

\* 3. 左旋操作有减少右子树高度的作用

\* 4. 同理，右旋操作有减少左子树高度的作用

\*/

void rb\_insert\_color(struct rb\_node \*node, struct rb\_root \*root)

{

struct rb\_node \*parent, \*gparent;

while ((parent = rb\_parent(node)) && rb\_is\_red(parent))

{

gparent = rb\_parent(parent);

/\* parent 是 gparent的左子树时 \*/

if (parent == gparent->rb\_left)

{

{

/\* gparent的左右子树的黑色节点都增加一个，仍然平衡 \*/

register struct rb\_node \*uncle = gparent->rb\_right;

if (uncle && rb\_is\_red(uncle))

{

rb\_set\_black(uncle);

rb\_set\_black(parent);

rb\_set\_red(gparent);

node = gparent;

continue;

}

}

/\* node为parent右子树时 \*/

if (parent->rb\_right == node)

{

register struct rb\_node \*tmp;

/\* 左旋后，parent的位置被node取代，然后再交换parent和node的位置，

\* 相当于node是parent的左子树

\* 由于node和parent都是红色（否则到不了这一步），parent左右子树的黑色节点数仍然是相等的

\*/

\_\_rb\_rotate\_left(parent, root);

tmp = parent;

parent = node;

node = tmp;

}

/\* parent 红->黑，gparent左子树比右子树多一个黑色节点

\* 右旋后，gparent左子树高度减一，减少的节点即parent，减少了一个黑色节点，parent变为新的gparent。

\* 所以右旋后，新的gparent的左右子树的黑色节点数再次平衡了

\*/

rb\_set\_black(parent);

rb\_set\_red(gparent);

\_\_rb\_rotate\_right(gparent, root);

/\* parent 是 gparent的右子树时，和上面的过程类似 \*/

} else {

{

register struct rb\_node \*uncle = gparent->rb\_left;

if (uncle && rb\_is\_red(uncle))

{

rb\_set\_black(uncle);

rb\_set\_black(parent);

rb\_set\_red(gparent);

node = gparent;

continue;

}

}

if (parent->rb\_left == node)

{

register struct rb\_node \*tmp;

\_\_rb\_rotate\_right(parent, root);

tmp = parent;

parent = node;

node = tmp;

}

rb\_set\_black(parent);

rb\_set\_red(gparent);

\_\_rb\_rotate\_left(gparent, root);

}

}

rb\_set\_black(root->rb\_node);

}

[复制代码](javascript:void(0);)

**函数4**：删除一个节点，并且调整删除后各节点的颜色。其中调整节点颜色其实是另一个单独的函数。

[复制代码](javascript:void(0);)

/\* 删除节点时，如果被删除的节点左子树==NULL或右子树==NULL或左右子树都==NULL

\* 那么只要把被删除节点的左子树或右子树直接关联到被删节点的父节点上即可，剩下的就是调整各节点颜色。

\* 只有被删节点是黑色才需要调整颜色，因为删除红色节点不影响红黑树的特性。

\*

\* 被删节点左右子树都存在的情况下，其实就是用中序遍历中被删节点的下一个节点来替代被删节点。

\* 代码中的操作只是将各个指针指向新的位置而已。

\*/

void rb\_erase(struct rb\_node \*node, struct rb\_root \*root)

{

struct rb\_node \*child, \*parent;

int color;

if (!node->rb\_left)

child = node->rb\_right;

else if (!node->rb\_right)

child = node->rb\_left;

else

{

struct rb\_node \*old = node, \*left;

/\* 寻找中序遍历中被删节点的下一个节点 \*/

node = node->rb\_right;

while ((left = node->rb\_left) != NULL)

node = left;

/\* 替换要删除的节点old \*/

if (rb\_parent(old)) {

if (rb\_parent(old)->rb\_left == old)

rb\_parent(old)->rb\_left = node;

else

rb\_parent(old)->rb\_right = node;

} else

root->rb\_node = node;

child = node->rb\_right;

parent = rb\_parent(node);

color = rb\_color(node);

if (parent == old) {

parent = node;

} else {

if (child)

rb\_set\_parent(child, parent);

parent->rb\_left = child;

node->rb\_right = old->rb\_right;

rb\_set\_parent(old->rb\_right, node);

}

node->rb\_parent\_color = old->rb\_parent\_color;

node->rb\_left = old->rb\_left;

rb\_set\_parent(old->rb\_left, node);

goto color;

}

parent = rb\_parent(node);

color = rb\_color(node);

if (child)

rb\_set\_parent(child, parent);

if (parent)

{

if (parent->rb\_left == node)

parent->rb\_left = child;

else

parent->rb\_right = child;

}

else

root->rb\_node = child;

color:

if (color == RB\_BLACK)

\_\_rb\_erase\_color(child, parent, root);

}

[复制代码](javascript:void(0);)

**函数5**：删除一个黑色节点后，重新调整相关节点的颜色。

[复制代码](javascript:void(0);)

/\* 这里的node就是上面函数中的child，所有node节点的左右子树肯定都是NULL

\* 不满足红黑树规则的就是从parent节点开始的子树，只要给从parent开始的子树增加一个黑色节点就行

\* 如果从parent节点开始的节点全是黑色，node和parent都继续向上移动

\*/

static void \_\_rb\_erase\_color(struct rb\_node \*node, struct rb\_node \*parent,

struct rb\_root \*root)

{

struct rb\_node \*other;

/\* （node不为NULL 且 node是黑色的） 或者 node == NULL \*/

while ((!node || rb\_is\_black(node)) && node != root->rb\_node)

{

if (parent->rb\_left == node)

{

other = parent->rb\_right;

if (rb\_is\_red(other))

{

rb\_set\_black(other);

rb\_set\_red(parent);

\_\_rb\_rotate\_left(parent, root);

other = parent->rb\_right;

}

/\* 如果从parent节点开始的节点全是黑色，node和parent都继续向上移动 \*/

if ((!other->rb\_left || rb\_is\_black(other->rb\_left)) &&

(!other->rb\_right || rb\_is\_black(other->rb\_right)))

{

rb\_set\_red(other);

node = parent;

parent = rb\_parent(node);

}

else

{

if (!other->rb\_right || rb\_is\_black(other->rb\_right))

{

rb\_set\_black(other->rb\_left);

rb\_set\_red(other);

\_\_rb\_rotate\_right(other, root);

other = parent->rb\_right;

}

rb\_set\_color(other, rb\_color(parent));

rb\_set\_black(parent);

rb\_set\_black(other->rb\_right);

\_\_rb\_rotate\_left(parent, root);

node = root->rb\_node;

break;

}

}

else

{

other = parent->rb\_left;

if (rb\_is\_red(other))

{

rb\_set\_black(other);

rb\_set\_red(parent);

\_\_rb\_rotate\_right(parent, root);

other = parent->rb\_left;

}

if ((!other->rb\_left || rb\_is\_black(other->rb\_left)) &&

(!other->rb\_right || rb\_is\_black(other->rb\_right)))

{

rb\_set\_red(other);

node = parent;

parent = rb\_parent(node);

}

else

{

if (!other->rb\_left || rb\_is\_black(other->rb\_left))

{

rb\_set\_black(other->rb\_right);

rb\_set\_red(other);

\_\_rb\_rotate\_left(other, root);

other = parent->rb\_left;

}

rb\_set\_color(other, rb\_color(parent));

rb\_set\_black(parent);

rb\_set\_black(other->rb\_left);

\_\_rb\_rotate\_right(parent, root);

node = root->rb\_node;

break;

}

}

}

if (node)

rb\_set\_black(node);

}

[复制代码](javascript:void(0);)

#### 4.3 使用示例

构造了一个内核模块来实际使用一下内核中的红黑树，代码在CentOS6.3 x64上运行通过。

C代码：

[复制代码](javascript:void(0);)

#include<linux/rbtree.h>

#include <linux/string.h>

#include "kn\_common.h"

MODULE\_LICENSE("Dual BSD/GPL");

struct student

{

int id;

char\* name;

struct rb\_node node;

};

static int insert\_student(struct student\*, struct rb\_root\*);

static int remove\_student(struct student\*, struct rb\_root\*);

static int display\_student(struct rb\_root\*, int);

static void display\_student\_from\_small(struct rb\_node\*);

static void display\_student\_from\_big(struct rb\_node\*);

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

static int testrbtree\_init(void)

{

#define N 10

struct rb\_root root = RB\_ROOT;

struct student \*stu[N];

char tmp\_name[5] = {'w', 'y', 'b', '0', '\0'};

int i;

// init N struct student

for (i=0; i<N; i++)

{

stu[i] = kmalloc(sizeof(struct student), GFP\_KERNEL);

stu[i]->id = i;

stu[i]->name = kmalloc(sizeof(char)\*5, GFP\_KERNEL);

tmp\_name[3] = (char)(i+48);

strcpy(stu[i]->name, tmp\_name);

// stu\_name[3] = (char)(i+48);

stu[i]->node.rb\_left = NULL;

stu[i]->node.rb\_right = NULL;

}

for (i=0; i < N; ++i)

{

printk(KERN\_ALERT "id=%d name=%s\n", stu[i]->id, stu[i]->name);

}

// add N student to rbtree

print\_current\_time(0);

for (i=0; i < N; i++)

insert\_student(stu[i], &root);

// display all students

printk(KERN\_ALERT "print from small to big!\n");

display\_student(&root, -1);

printk(KERN\_ALERT "print from big to small!\n");

display\_student(&root, 1);

// delete student 8

remove\_student(stu[7], &root);

display\_student(&root, -1);

// free all student

for (i=0; i<N; ++i)

{

kfree(stu[i]->name);

kfree(stu[i]);

}

return 0;

}

static int insert\_student(struct student\* stu, struct rb\_root\* root)

{

struct rb\_node\* parent;

struct rb\_node\* tmp\_rb;

struct student\* tmp\_stu;

/\* first time to insert node \*/

if (!root->rb\_node)

{

root->rb\_node = &(stu->node);

rb\_set\_parent(&(stu->node), NULL);

rb\_set\_black(&(stu->node));

return 0;

}

/\* find where to insert node \*/

tmp\_rb = root->rb\_node;

while(tmp\_rb)

{

parent = tmp\_rb;

tmp\_stu = rb\_entry(tmp\_rb, struct student, node);

if (tmp\_stu->id > stu->id)

tmp\_rb = parent->rb\_left;

else if (tmp\_stu->id < stu->id)

tmp\_rb = parent->rb\_right;

else

break;

}

/\* the student's id is already in the rbtree \*/

if (tmp\_rb)

{

printk(KERN\_ALERT "this student has been inserted!\n");

return 1;

}

if (tmp\_stu->id > stu->id)

parent->rb\_left = &(stu->node);

else

parent->rb\_right = &(stu->node);

rb\_set\_parent(&(stu->node), parent);

rb\_insert\_color(&(stu->node), root);

return 0;

}

static int remove\_student(struct student\* stu, struct rb\_root\* root)

{

rb\_erase(&(stu->node), root);

return 0;

}

static int display\_student(struct rb\_root \*root, int order)

{

if (!root->rb\_node)

return 1;

if (order < 0)

display\_student\_from\_small(root->rb\_node);

else

display\_student\_from\_big(root->rb\_node);

return 0;

}

static void display\_student\_from\_small(struct rb\_node\* node)

{

struct student \*tmp\_stu;

if (node)

{

display\_student\_from\_small(node->rb\_left);

tmp\_stu = rb\_entry(node, struct student, node);

print\_student(tmp\_stu);

display\_student\_from\_small(node->rb\_right);

}

}

static void display\_student\_from\_big(struct rb\_node\* node)

{

struct student \*tmp\_stu;

if (node)

{

display\_student\_from\_big(node->rb\_right);

tmp\_stu = rb\_entry(node, struct student, node);

print\_student(tmp\_stu);

display\_student\_from\_big(node->rb\_left);

}

}

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

{

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

print\_current\_time(0);

printk(KERN\_ALERT "id=%d\tname=%s\n", stu->id, stu->name);

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

}

static void testrbtree\_exit(void)

{

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

print\_current\_time(0);

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

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

}

module\_init(testrbtree\_init);

module\_exit(testrbtree\_exit);

[复制代码](javascript:void(0);)

***注：***其中用到的kn\_common.h和kn\_common.c文件与队列的示例中一样。

Makefile：

[复制代码](javascript:void(0);)

obj-m += rbtree.o

rbtree-objs := testrbtree.o kn\_common.o

#generate the path

CURRENT\_PATH:=$(shell pwd)

#the current kernel version number

LINUX\_KERNEL:=$(shell uname -r)

#the absolute path

LINUX\_KERNEL\_PATH:=/usr/src/kernels/$(LINUX\_KERNEL)

#complie object

all:

make -C $(LINUX\_KERNEL\_PATH) M=$(CURRENT\_PATH) modules

rm -rf modules.order Module.symvers .\*.cmd \*.o \*.mod.c .tmp\_versions \*.unsigned

#clean

clean:

rm -rf modules.order Module.symvers .\*.cmd \*.o \*.mod.c \*.ko .tmp\_versions \*.unsigned

[复制代码](javascript:void(0);)

安装,卸载内核模块以及查看内核模块的运行结果：

insmod rbtree.ko

rmmod rbtree

dmesg | tail -135

# [中断处理](http://www.cnblogs.com/wang_yb/archive/2013/04/19/3030345.html)

中断处理一般不是纯软件来实现的，需要硬件的支持。通过对中断的学习有助于更深入的了解系统的一些底层原理，特别是驱动程序的开发。

主要内容：

* 什么是中断
* 中断类型
* 中断相关函数
* 中断处理机制
* 中断控制方法
* 总结

## 什么是中断

为了提高CPU和外围硬件(硬盘，键盘，鼠标等等)之间协同工作的性能，引入了中断的机制。

没有中断的话，CPU和外围设备之间协同工作可能只有轮询这个方法：CPU定期检查硬件状态，需要处理时就处理，否则就跳过。

当硬件忙碌的时候，CPU很可能会做许多无用功（每次轮询都是跳过不处理）。

中断机制是硬件在需要的时候向CPU发出信号，CPU暂时停止正在进行的工作，来处理硬件请求的一种机制。

## 中断类型

中断一般分为异步中断(一般由硬件引起)和同步中断(一般由处理器本身引起)。

异步中断：CPU处理中断的时间过长，所以先将硬件复位，使硬件可以继续自己的工作，然后在适当时候处理中断请求中耗时的部分。

举个例子：网卡的工作原理

1. 网卡收到数据包后，向CPU发出中断信号，请求处理接收到的数据包
2. CPU将收到的数据包拷贝到内存后，即通知网卡继续工作
3. 至于数据包拷贝至内存后的处理会在适当的时候进行（任务异步执行）

 这样做避免了处理数据包时间过长导致网卡接收数据包速度变慢。

同步中断：CPU处理完中断请求的所有工作后才反馈硬件

举个例子：系统异常处理(比如运算中的除0操作)

1. 应用程序出现异常后，需要内核来处理
2. 内核调用相应的异常处理函数来处理异常
3. 处理完后终了应用程序或者给出message

同步中断应该处理能很快完成的一种中断。

## 中断相关函数

实现一个中断，主要需要知道3个函数：

* 注册中断的函数
* 释放中断的函数
* 中断处理程序的声明

### 注册中断的函数

    位置：<linux/interrupt.h>  include/linux/interrupt.h

定义如下：

[复制代码](javascript:void(0);)

/\*

\* irg - 表示要分配的中断号

\* handler - 实际的中断处理程序

\* flags - 标志位，表示此中断的具有特性

\* name - 中断设备名称的ASCII 表示，这些会被/proc/irq和/proc/interrupts文件使用

\* dev - 用于共享中断线，多个中断程序共享一个中断线时(共用一个中断号)，依靠dev来区别各个中断程序

\* 返回值：

\* 执行成功：0

\* 执行失败：非0

\*/

int request\_irq(unsigned int irq,

irq\_handler\_t handler,

unsigned long flags,

const char\* name,

void \*dev)

[复制代码](javascript:void(0);)

### 释放中断的函数

定义比较简单：

void free\_irq(unsigned int irq, void \*dev)

如果不是共享中断线，则直接删除irq对应的中断线。

如果是共享中断线，则判断此中断处理程序是否中断线上的最后一个中断处理程序，

    是最后一个中断处理程序 -> 删除中断线和中断处理程序

    不是最后一个中断处理程序 -> 删除中断处理程序

### 中断处理程序的声明

声明格式如下：

[复制代码](javascript:void(0);)

/\*

\* 中断处理程序的声明

\* @irp - 中断处理程序(即request\_irq()中handler)关联的中断号

\* @dev - 与 request\_irq()中的dev一样，表示一个设备的结构体

\* 返回值：

\* irqreturn\_t - 执行成功：IRQ\_HANDLED 执行失败：IRQ\_NONE

\*/

static irqreturn\_t intr\_handler(int, irq, void \*dev)

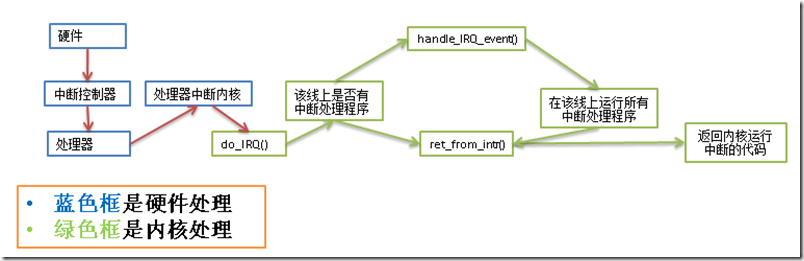
[复制代码](javascript:void(0);)

## 中断处理机制

中断处理的过程主要涉及3函数：

* do\_IRQ 与体系结构有关，对所接收的中断进行应答
* handle\_IRQ\_event 调用中断线上所有中断处理
* ret\_from\_intr 恢复寄存器，将内核恢复到中断前的状态

处理流程可以参见书中的图，如下：

[](https://images0.cnblogs.com/blog/83005/201304/19113723-21463cd0f94f455480d282ac1b38138d.png)

## 中断控制方法

常用的中断控制方法见下表：

|  |  |
| --- | --- |
| **函数** | **说明** |
| local\_irq\_disable() | 禁止本地中断传递 |
| local\_irq\_enable() | 激活本地中断传递 |
| local\_irq\_save() | 保存本地中断传递的当前状态，然后禁止本地中断传递 |
| local\_irq\_restore() | 恢复本地中断传递到给定的状态 |
| disable\_irq() | 禁止给定中断线，并确保该函数返回之前在该中断线上没有处理程序在运行 |
| disable\_irq\_nosync() | 禁止给定中断线 |
| enable\_irq() | 激活给定中断线 |
| irqs\_disabled() | 如果本地中断传递被禁止，则返回非0；否则返回0 |
| in\_interrupt() | 如果在中断上下文中，则返回非0；如果在进程上下文中，则返回0 |
| in\_irq() | 如果当前正在执行中断处理程序，则返回非0；否则返回0 |

## 总结

中断处理对处理时间的要求很高，如果一个中断要花费较长时间，那么中断处理一般分为2部分。上半部只做一些必要的工作后，立即通知硬件继续自己的工作。中断处理中耗时的部分，也就是下半部的工作，CPU会在适当的时候去完成。

# [中断下半部的处理](http://www.cnblogs.com/wang_yb/archive/2013/04/23/3037268.html)

在前一章也提到过，之所以中断会分成上下两部分，是由于中断对时限的要求非常高，需要尽快的响应硬件。

**主要内容：**

* 中断下半部处理
* 实现中断下半部的机制
* 总结中断下半部的实现
* 中断实现示例

## 中断下半部处理

那么对于一个中断，如何划分上下两部分呢？哪些处理放在上半部，哪些处理放在下半部？

这里有一些经验可供借鉴：

1. 如果一个任务对时间十分敏感，将其放在上半部
2. 如果一个任务和硬件有关，将其放在上半部
3. 如果一个任务要保证不被其他中断打断，将其放在上半部
4. 其他所有任务，考虑放在下半部

## 实现中断下半部的机制

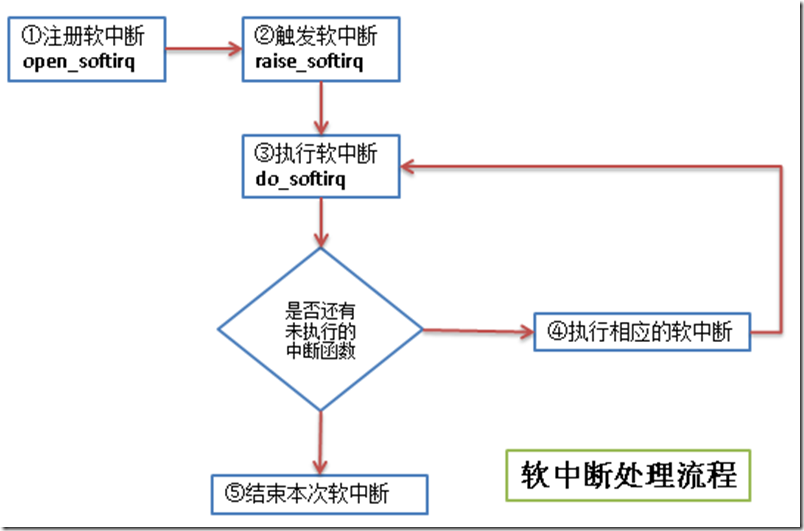
实现下半部的方法很多，随着内核的发展，产生了一些新的方法，也淘汰了一些旧方法。目前使用最多的是以下3中方法

* 2.1 软中断
* 2.2 tasklet
* 2.3 工作队列

### 软中断

软中断的代码在：kernel/softirq.c

软中断的流程如下：

[](https://images0.cnblogs.com/blog/83005/201304/23111716-dc0f4c90540c48569dd18f835b2d8af9.png)

流程图中几个步骤的说明：

① 注册软中断的函数 open\_softirq参见 kernel/softirq.c文件)

[复制代码](javascript:void(0);)

/\*

\* 将软中断类型和软中断处理函数加入到软中断序列中

\* @nr - 软中断类型

\* @(\*action)(struct softirq\_action \*) - 软中断处理的函数指针

\*/

void open\_softirq(int nr, void (\*action)(struct softirq\_action \*))

{

/\* softirq\_vec是个struct softirq\_action类型的数组 \*/

softirq\_vec[nr].action = action;

}

[复制代码](javascript:void(0);)

软中断类型目前有10个，其定义在 include/linux/interrupt.h 文件中：

[复制代码](javascript:void(0);)

enum

{

HI\_SOFTIRQ=0,

TIMER\_SOFTIRQ,

NET\_TX\_SOFTIRQ,

NET\_RX\_SOFTIRQ,

BLOCK\_SOFTIRQ,

BLOCK\_IOPOLL\_SOFTIRQ,

TASKLET\_SOFTIRQ,

SCHED\_SOFTIRQ,

HRTIMER\_SOFTIRQ,

RCU\_SOFTIRQ, /\* Preferable RCU should always be the last softirq \*/

NR\_SOFTIRQS

};

[复制代码](javascript:void(0);)

struct softirq\_action 的定义也在 include/linux/interrupt.h 文件中

[复制代码](javascript:void(0);)

/\*

\* 这个结构体的字段是个函数指针，字段名称是action

\* 函数指针的返回指是void型

\* 函数指针的参数是 struct softirq\_action 的地址，其实就是指向 softirq\_vec 中的某一项

\* 如果 open\_softirq 是这样调用的： open\_softirq(NET\_TX\_SOFTIRQ, my\_tx\_action);

\* 那么 my\_tx\_action 的参数就是 softirq\_vec[NET\_TX\_SOFTIRQ]的地址

\*/

struct softirq\_action

{

void (\*action)(struct softirq\_action \*);

};

[复制代码](javascript:void(0);)

② 触发软中断的函数 raise\_softirq 参见 kernel/softirq.c文件

[复制代码](javascript:void(0);)

/\*

\* 触发某个中断类型的软中断

\* @nr - 被触发的中断类型

\* 从函数中可以看出，在处理软中断前后有保存和恢复寄存器的操作

\*/

void raise\_softirq(unsigned int nr)

{

unsigned long flags;

local\_irq\_save(flags);

raise\_softirq\_irqoff(nr);

local\_irq\_restore(flags);

}

[复制代码](javascript:void(0);)

③ 执行软中断 do\_softirq 参见 kernel/softirq.c文件

[复制代码](javascript:void(0);)

asmlinkage void do\_softirq(void)

{

\_\_u32 pending;

unsigned long flags;

/\* 判断是否在中断处理中，如果正在中断处理，就直接返回 \*/

if (in\_interrupt())

return;

/\* 保存当前寄存器的值 \*/

local\_irq\_save(flags);

/\* 取得当前已注册软中断的位图 \*/

pending = local\_softirq\_pending();

/\* 循环处理所有已注册的软中断 \*/

if (pending)

\_\_do\_softirq();

/\* 恢复寄存器的值到中断处理前 \*/

local\_irq\_restore(flags);

}

[复制代码](javascript:void(0);)

④ 执行相应的软中断 - 执行自己写的中断处理

linux中，执行软中断有专门的内核线程，每个处理器对应一个线程，名称ksoftirqd/n (n对应处理器号)

通过top命令查看我的单核虚拟机，CentOS系统中的ksoftirqd线程如下：

[root@vbox ~]# top | grep ksoftirq

4 root 20 0 0 0 0 S 0.0 0.0 0:00.02 ksoftirqd/0

### tasklet

tasklet也是利用软中断来实现的，但是它提供了比软中断更好用的接口(其实就是基于软中断又封装了一下)，所以除了对性能要求特别高的情况，一般建议使用tasklet来实现自己的中断。

tasklet对应的结构体在 <linux/interrupt.h> 中

[复制代码](javascript:void(0);)

struct tasklet\_struct

{

struct tasklet\_struct \*next; /\* 链表中的下一个tasklet \*/

unsigned long state; /\* tasklet状态 \*/

atomic\_t count; /\* 引用计数器 \*/

void (\*func)(unsigned long); /\* tasklet处理函数 \*/

unsigned long data; /\* tasklet处理函数的参数 \*/

};

[复制代码](javascript:void(0);)

tasklet状态只有3种值：

1. 值 0 表示该tasklet没有被调度
2. 值 TASKLET\_STATE\_SCHED 表示该tasklet已经被调度
3. 值 TASKLET\_STATE\_RUN 表示该tasklet已经运行

引用计数器count 的值不为0，表示该tasklet被禁止。

tasklet使用流程如下：

1. 声明tasklet (参见<linux/interrupt.h>)

[复制代码](javascript:void(0);)

/\* 静态声明一个tasklet \*/

#define DECLARE\_TASKLET(name, func, data) \

struct tasklet\_struct name = { NULL, 0, ATOMIC\_INIT(0), func, data }

#define DECLARE\_TASKLET\_DISABLED(name, func, data) \

struct tasklet\_struct name = { NULL, 0, ATOMIC\_INIT(1), func, data }

/\* 动态声明一个tasklet 传递一个tasklet\_struct指针给初始化函数 \*/

extern void tasklet\_init(struct tasklet\_struct \*t,

void (\*func)(unsigned long), unsigned long data);

[复制代码](javascript:void(0);)

2. 编写处理程序

参照tasklet处理函数的原型来写自己的处理逻辑

void tasklet\_handler(unsigned long date)

3. 调度tasklet

中断的上半部处理完后调度tasklet，在适当时候进行下半部的处理

tasklet\_schedule(&my\_tasklet) /\* my\_tasklet就是之前声明的tasklet\_struct \*/

### 工作队列

工作队列子系统是一个用于创建内核线程的接口，通过它可以创建一个工作者线程来专门处理中断的下半部工作。

工作队列和tasklet不一样，不是基于软中断来实现的。

缺省的工作者线程名称是 events/n (n对应处理器号)。

通过top命令查看我的单核虚拟机，CentOS系统中的events线程如下：

[root@vbox ~]# top | grep event

7 root 20 0 0 0 0 S 0.0 0.0 0:03.71 events/0

工作队列主要用到下面3个结构体，弄懂了这3个结构体的关系，也就知道工作队列的处理流程了。

[复制代码](javascript:void(0);)

/\* 在 include/linux/workqueue.h 文件中定义 \*/

struct work\_struct {

atomic\_long\_t data; /\* 这个并不是处理函数的参数，而是表示此work是否pending等状态的flag \*/

#define WORK\_STRUCT\_PENDING 0 /\* T if work item pending execution \*/

#define WORK\_STRUCT\_FLAG\_MASK (3UL)

#define WORK\_STRUCT\_WQ\_DATA\_MASK (~WORK\_STRUCT\_FLAG\_MASK)

struct list\_head entry; /\* 中断下半部处理函数的链表 \*/

work\_func\_t func; /\* 处理中断下半部工作的函数 \*/

#ifdef CONFIG\_LOCKDEP

struct lockdep\_map lockdep\_map;

#endif

};

/\* 在 kernel/workqueue.c文件中定义

\* 每个工作者线程对应一个 cpu\_workqueue\_struct ，其中包含要处理的工作的链表

\* (即 work\_struct 的链表，当此链表不空时，唤醒工作者线程来进行处理)

\*/

/\*

\* The per-CPU workqueue (if single thread, we always use the first

\* possible cpu).

\*/

struct cpu\_workqueue\_struct {

spinlock\_t lock; /\* 锁保护这种结构 \*/

struct list\_head worklist; /\* 工作队列头节点 \*/

wait\_queue\_head\_t more\_work;

struct work\_struct \*current\_work;

struct workqueue\_struct \*wq; /\* 关联工作队列结构 \*/

struct task\_struct \*thread; /\* 关联线程 \*/

} \_\_\_\_cacheline\_aligned;

/\* 也是在 kernel/workqueue.c 文件中定义的

\* 每个 workqueue\_struct 表示一种工作者类型，系统默认的就是 events 工作者类型

\* 每个工作者类型一般对应n个工作者线程，n就是处理器的个数

\*/

/\*

\* The externally visible workqueue abstraction is an array of

\* per-CPU workqueues:

\*/

struct workqueue\_struct {

struct cpu\_workqueue\_struct \*cpu\_wq; /\* 工作者线程 \*/

struct list\_head list;

const char \*name;

int singlethread;

int freezeable; /\* Freeze threads during suspend \*/

int rt;

#ifdef CONFIG\_LOCKDEP

struct lockdep\_map lockdep\_map;

#endif

};

[复制代码](javascript:void(0);)

使用工作者队列的方法见下图：

[](https://images0.cnblogs.com/blog/83005/201304/23111723-ae64659dc99a4c708c62357e5c403b51.png)

① 创建推后执行的工作 - 有静态创建和动态创建2种方法

[复制代码](javascript:void(0);)

/\* 静态创建一个work\_struct

\* @n - work\_struct结构体，不用事先定义

\* @f - 下半部处理函数

\*/

#define DECLARE\_WORK(n, f) \

struct work\_struct n = \_\_WORK\_INITIALIZER(n, f)

/\* 动态创建一个 work\_struct

\* @\_work - 已经定义好的一个 work\_struct

\* @\_func - 下半部处理函数

\*/

#ifdef CONFIG\_LOCKDEP

#define INIT\_WORK(\_work, \_func) \

do { \

static struct lock\_class\_key \_\_key; \

\

(\_work)->data = (atomic\_long\_t) WORK\_DATA\_INIT(); \

lockdep\_init\_map(&(\_work)->lockdep\_map, #\_work, &\_\_key, 0);\

INIT\_LIST\_HEAD(&(\_work)->entry); \

PREPARE\_WORK((\_work), (\_func)); \

} while (0)

#else

#define INIT\_WORK(\_work, \_func) \

do { \

(\_work)->data = (atomic\_long\_t) WORK\_DATA\_INIT(); \

INIT\_LIST\_HEAD(&(\_work)->entry); \

PREPARE\_WORK((\_work), (\_func)); \

} while (0)

#endif

[复制代码](javascript:void(0);)

工作队列处理函数的原型：

typedef void (\*work\_func\_t)(struct work\_struct \*work);

② 刷新现有的工作，这个步骤不是必须的，可以直接从第①步直接进入第③步

   刷新现有工作的意思就是在追加新的工作之前，保证队列中的已有工作已经执行完了。

[复制代码](javascript:void(0);)

/\* 刷新系统默认的队列，即 events 队列 \*/

void flush\_scheduled\_work(void);

/\* 刷新用户自定义的队列

\* @wq - 用户自定义的队列

\*/

void flush\_workqueue(struct workqueue\_struct \*wq);

[复制代码](javascript:void(0);)

③ 调度工作 - 调度新定义的工作，使之处于等待处理器执行的状态

[复制代码](javascript:void(0);)

/\* 调度第一步中新定义的工作，在系统默认的工作者线程中执行此工作

\* @work - 第一步中定义的工作

\*/

schedule\_work(struct work\_struct \*work);

/\* 调度第一步中新定义的工作，在系统默认的工作者线程中执行此工作

\* @work - 第一步中定义的工作

\* @delay - 延迟的时钟节拍

\*/

int schedule\_delayed\_work(struct delayed\_work \*work, unsigned long delay);

/\* 调度第一步中新定义的工作，在用户自定义的工作者线程中执行此工作

\* @wq - 用户自定义的工作队列类型

\* @work - 第一步中定义的工作

\*/

int queue\_work(struct workqueue\_struct \*wq, struct work\_struct \*work);

/\* 调度第一步中新定义的工作，在用户自定义的工作者线程中执行此工作

\* @wq - 用户自定义的工作队列类型

\* @work - 第一步中定义的工作

\* @delay - 延迟的时钟节拍

\*/

int queue\_delayed\_work(struct workqueue\_struct \*wq,

struct delayed\_work \*work, unsigned long delay);

[复制代码](javascript:void(0);)

## 总结中断下半部的实现

下面对实现中断下半部工作的3种机制进行总结，便于在实际使用中决定使用哪种机制

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **下半部机制** | **上下文** | **复杂度** | **执行性能** | **顺序执行保障** |
| 软中断 | 中断 | 高  (需要自己确保软中断的执行顺序及锁机制) | 好  (全部自己实现，便于调优) | 没有 |
| tasklet | 中断 | 中  (提供了简单的接口来使用软中断) | 中 | 同类型不能同时执行 |
| 工作队列 | 进程 | 低  (在进程上下文中运行，与写用户程序差不多) | 差 | 没有  (和进程上下文一样被调度) |

## 中断实现示例

#### 4.1 软中断的实现

本来想用内核模块的方法来测试一下软中断的流程，但是编译时发现软中断注册函数(open\_softirq)和触发函数(raise\_softirq)

并没有用EXPORT\_SYMBOL导出，所以自定义的内核模块中无法使用。

测试的代码如下：

[复制代码](javascript:void(0);)

#include <linux/interrupt.h>

#include "kn\_common.h"

MODULE\_LICENSE("Dual BSD/GPL");

static void my\_softirq\_func(struct softirq\_action\*);

static int testsoftirq\_init(void)

{

// 注册softirq，这里注册的是定时器的下半部

open\_softirq(TIMER\_SOFTIRQ, my\_softirq\_func);

// 触发softirq

raise\_softirq(TIMER\_SOFTIRQ);

return 0;

}

static void testsoftirq\_exit(void)

{

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

print\_current\_time(0);

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

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

}

static void my\_softirq\_func(struct softirq\_action\* act)

{

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

print\_current\_time(0);

printk(KERN\_ALERT "my softirq function is been called!....\n");

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

}

module\_init(testsoftirq\_init);

module\_exit(testsoftirq\_exit);

[复制代码](javascript:void(0);)

其中头文件 **kn\_common.h** 的相关内容参见之前的博客[《Linux内核设计与实现》读书笔记（六）- 内核数据结构](http://www.cnblogs.com/wang_yb/archive/2013/04/16/3023892.html" \t "_blank)

由于内核没有用**EXPORT\_SYMBOL**导出**open\_softirq**和**raise\_softirq**函数，所以编译时有如下警告：

WARNING: "open\_softirq" [/root/chap08/mysoftirq.ko] undefined!

WARNING: "raise\_softirq" [/root/chap08/mysoftirq.ko] undefined!

**注：**编译用的系统时centos6.3 (uname -r结果 - 2.6.32-279.el6.x86\_64)

没办法，只能尝试修改内核代码(将open\_softirq和raise\_softirq用EXPORT\_SYMBOL导出)，再重新编译内核，然后再尝试能否测试软中断。

主要修改2个文件，(既然要修改代码，干脆加了一种软中断类型)：

[复制代码](javascript:void(0);)

/\* 修改 kernel/softirq.c \*/

// ... 略 ...

char \*softirq\_to\_name[NR\_SOFTIRQS] = {

"HI", "TIMER", "NET\_TX", "NET\_RX", "BLOCK", "BLOCK\_IOPOLL",

"TASKLET", "SCHED", "HRTIMER", "RCU", "WYB"

}; /\* 追加了一种新的softirq，即 "WYB"，我名字的缩写 ^\_^ \*/

// ... 略 ...

void raise\_softirq(unsigned int nr)

{

unsigned long flags;

local\_irq\_save(flags);

raise\_softirq\_irqoff(nr);

local\_irq\_restore(flags);

}

EXPORT\_SYMBOL(raise\_softirq); /\* 追加的代码 \*/

void open\_softirq(int nr, void (\*action)(struct softirq\_action \*))

{

softirq\_vec[nr].action = action;

}

EXPORT\_SYMBOL(open\_softirq); /\* 追加的代码 \*/

// ... 略 ...

/\* 还修改了 include/linux/interrupt.h \*/

enum

{

HI\_SOFTIRQ=0,

TIMER\_SOFTIRQ,

NET\_TX\_SOFTIRQ,

NET\_RX\_SOFTIRQ,

BLOCK\_SOFTIRQ,

BLOCK\_IOPOLL\_SOFTIRQ,

TASKLET\_SOFTIRQ,

SCHED\_SOFTIRQ,

HRTIMER\_SOFTIRQ,

RCU\_SOFTIRQ, /\* Preferable RCU should always be the last softirq \*/

WYB\_SOFTIRQS, /\* 追加的一种中断类型 \*/

NR\_SOFTIRQS

};

[复制代码](javascript:void(0);)

重新编译内核后，在新的内核上再次实验软中断代码：

(编译内核方法参见：[《Linux内核设计与实现》读书笔记（五）- 系统调用](http://www.cnblogs.com/wang_yb/archive/2012/09/17/2688263.html" \t "_blank) 3.3节)

测试软中断的代码：testsoftirq.c

[复制代码](javascript:void(0);)

#include <linux/interrupt.h>

#include "kn\_common.h"

MODULE\_LICENSE("Dual BSD/GPL");

static void my\_softirq\_func(struct softirq\_action\*);

static int testsoftirq\_init(void)

{

printk(KERN\_ALERT "interrupt's top half!\n");

// 注册softirq，这里注册的是自定义的软中断类型

open\_softirq(WYB\_SOFTIRQS, my\_softirq\_func);

// 触发softirq

raise\_softirq(WYB\_SOFTIRQS);

return 0;

}

static void testsoftirq\_exit(void)

{

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

print\_current\_time(0);

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

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

}

static void my\_softirq\_func(struct softirq\_action\* act)

{

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

print\_current\_time(0);

printk(KERN\_ALERT "my softirq function is been called!....\n");

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

}

module\_init(testsoftirq\_init);

module\_exit(testsoftirq\_exit);

[复制代码](javascript:void(0);)

Makefile：

[复制代码](javascript:void(0);)

obj-m += mysoftirq.o

mysoftirq-objs := testsoftirq.o kn\_common.o

#generate the path

CURRENT\_PATH:=$(shell pwd)

#the current kernel version number

LINUX\_KERNEL:=$(shell uname -r)

#the absolute path

LINUX\_KERNEL\_PATH:=/usr/src/kernels/$(LINUX\_KERNEL)

#complie object

all:

make -C $(LINUX\_KERNEL\_PATH) M=$(CURRENT\_PATH) modules

rm -rf modules.order Module.symvers .\*.cmd \*.o \*.mod.c .tmp\_versions \*.unsigned

#clean

clean:

rm -rf modules.order Module.symvers .\*.cmd \*.o \*.mod.c \*.ko .tmp\_versions \*.unsigned

[复制代码](javascript:void(0);)

测试软中断的方法如下：

[复制代码](javascript:void(0);)

make

insmod mysoftirq.ko

rmmod mysoftirq

dmesg | tail -9

# 运行结果

interrupt's top half!

=========================

2013-4-22 14:4:57

my softirq function is been called!....

=========================

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

2013-4-22 14:5:2

testsoftirq is exited!

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

[复制代码](javascript:void(0);)

#### 4.2 tasklet的实现

tasklet的实验用默认的内核即可，我们切换到centos6.3的默认内核(uname -r: 2.6.32-279.el6.x86\_64)

从中我们也可以看出，内核之所以没有导出open\_softirq和raise\_softirq函数，可能还是因为提倡我们尽量用tasklet来实现中断的下半部工作。

tasklet测试代码：testtasklet.c

[复制代码](javascript:void(0);)

#include <linux/interrupt.h>

#include "kn\_common.h"

MODULE\_LICENSE("Dual BSD/GPL");

static void my\_tasklet\_func(unsigned long);

/\* mytasklet 必须定义在testtasklet\_init函数的外面，否则会出错 \*/

DECLARE\_TASKLET(mytasklet, my\_tasklet\_func, 1000);

static int testtasklet\_init(void)

{

printk(KERN\_ALERT "interrupt's top half!\n");

// 如果在这里定义的话，那么 mytasklet是函数的局部变量，

// 后面调度的时候会找不到 mytasklet

// DECLARE\_TASKLET(mytasklet, my\_tasklet\_func, 1000);

// 调度tasklet， 处理器会在适当时候执行这个tasklet

tasklet\_schedule(&mytasklet);

return 0;

}

static void testtasklet\_exit(void)

{

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

print\_current\_time(0);

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

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

}

static void my\_tasklet\_func(unsigned long data)

{

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

print\_current\_time(0);

printk(KERN\_ALERT "my tasklet function is been called!....\n");

printk(KERN\_ALERT "parameter data is %ld\n", data);

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

}

module\_init(testtasklet\_init);

module\_exit(testtasklet\_exit);

[复制代码](javascript:void(0);)

Makefile：

[复制代码](javascript:void(0);)

obj-m += mytasklet.o

mytasklet-objs := testtasklet.o kn\_common.o

#generate the path

CURRENT\_PATH:=$(shell pwd)

#the current kernel version number

LINUX\_KERNEL:=$(shell uname -r)

#the absolute path

LINUX\_KERNEL\_PATH:=/usr/src/kernels/$(LINUX\_KERNEL)

#complie object

all:

make -C $(LINUX\_KERNEL\_PATH) M=$(CURRENT\_PATH) modules

rm -rf modules.order Module.symvers .\*.cmd \*.o \*.mod.c .tmp\_versions \*.unsigned

#clean

clean:

rm -rf modules.order Module.symvers .\*.cmd \*.o \*.mod.c \*.ko .tmp\_versions \*.unsigned

[复制代码](javascript:void(0);)

测试tasklet的方法如下：

[复制代码](javascript:void(0);)

make

insmod mytasklet.ko

rmmod mytasklet

dmesg | tail -10

# 运行结果

interrupt's top half!

=========================

2013-4-22 14:53:14

my tasklet function is been called!....

parameter data is 1000

=========================

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

2013-4-22 14:53:20

testtasklet is exited!

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

[复制代码](javascript:void(0);)

### 工作队列的实现

workqueue的例子的中静态定义了一个工作，动态定义了一个工作。

静态定义的工作由系统工作队列（events/n）调度，

动态定义的工作由自定义的工作队列（myworkqueue）调度。

测试工作队列的代码：testworkqueue.c

[复制代码](javascript:void(0);)

#include <linux/workqueue.h>

#include "kn\_common.h"

MODULE\_LICENSE("Dual BSD/GPL");

static void my\_work\_func(struct work\_struct \*);

static void my\_custom\_workqueue\_func(struct work\_struct \*);

/\* 静态创建一个工作，使用系统默认的工作者线程，即 events/n \*/

DECLARE\_WORK(mywork, my\_work\_func);

static int testworkqueue\_init(void)

{

/\*自定义的workqueue \*/

struct workqueue\_struct \*myworkqueue = create\_workqueue("myworkqueue");

/\* 动态创建一个工作 \*/

struct work\_struct \*mywork2;

mywork2 = kmalloc(sizeof(struct work\_struct), GFP\_KERNEL);

INIT\_WORK(mywork2, my\_custom\_workqueue\_func);

printk(KERN\_ALERT "interrupt's top half!\n");

/\* 刷新系统默认的队列 \*/

flush\_scheduled\_work();

/\* 调度工作 \*/

schedule\_work(&mywork);

/\* 刷新自定义的工作队列 \*/

flush\_workqueue(myworkqueue);

/\* 调度自定义工作队列上的工作 \*/

queue\_work(myworkqueue, mywork2);

return 0;

}

static void testworkqueue\_exit(void)

{

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

print\_current\_time(0);

printk(KERN\_ALERT "my workqueue test is exited!\n");

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

}

static void my\_work\_func(struct work\_struct \*work)

{

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

print\_current\_time(0);

printk(KERN\_ALERT "my workqueue function is been called!....\n");

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

}

static void my\_custom\_workqueue\_func(struct work\_struct \*work)

{

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

print\_current\_time(0);

printk(KERN\_ALERT "my cutomize workqueue function is been called!....\n");

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

kfree(work);

}

module\_init(testworkqueue\_init);

module\_exit(testworkqueue\_exit);

[复制代码](javascript:void(0);)

Makefile:

[复制代码](javascript:void(0);)

obj-m += myworkqueue.o

myworkqueue-objs := testworkqueue.o kn\_common.o

#generate the path

CURRENT\_PATH:=$(shell pwd)

#the current kernel version number

LINUX\_KERNEL:=$(shell uname -r)

#the absolute path

LINUX\_KERNEL\_PATH:=/usr/src/kernels/$(LINUX\_KERNEL)

#complie object

all:

make -C $(LINUX\_KERNEL\_PATH) M=$(CURRENT\_PATH) modules

rm -rf modules.order Module.symvers .\*.cmd \*.o \*.mod.c .tmp\_versions \*.unsigned

#clean

clean:

rm -rf modules.order Module.symvers .\*.cmd \*.o \*.mod.c \*.ko .tmp\_versions \*.unsigned

[复制代码](javascript:void(0);)

测试workqueue的方法如下：

[复制代码](javascript:void(0);)

make

insmod myworkqueue.ko

rmmod myworkqueue

dmesg | tail -13

# 运行结果

interrupt's top half!

=========================

2013-4-23 9:55:29

my workqueue function is been called!....

=========================

=========================

2013-4-23 9:55:29

my cutomize workqueue function is been called!....

=========================

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

2013-4-23 9:55:29

my workqueue is exited!

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# [（九）- 内核同步介绍](http://www.cnblogs.com/wang_yb/archive/2013/04/24/3040712.html)

存在共享资源(共享一个文件，一块内存等等)的时候，为了防止并发访问时共享资源的数据不一致，引入了同步机制。

**主要内容：**

1. 同步的概念
2. 同步的方法-加锁
3. 死锁
4. 锁的粒度

### 1. 同步的概念

了解同步之前，先了解另外2个概念：

* 临界区   - 也称为临界段，就是访问和操作共享数据的代码段。
* 竞争条件 - 2个或2个以上线程在临界区里同时执行的时候，就构成了竞争条件。

所谓同步，其实防止在临界区中形成竞争条件。

如果临界区里是原子操作(即整个操作完成前不会被打断)，那么自然就不会出竞争条件。

但在实际应用中，临界区中的代码往往不会那么简单，所以为了保持同步，引入了锁机制。

### 2. 同步的方法-加锁

为了给临界区加锁，保证临界区数据的同步，首先了解一下内核中哪些情况下会产生并发。

内核中造成竞争条件的原因：

|  |  |
| --- | --- |
| **竞争原因** | **说明** |
| 中断 | 中断随时会发生，也就会随时打断当前执行的代码。如果中断和被打断的代码在相同的临界区，就产生了竞争条件 |
| 软中断和tasklet | 软中断和tasklet也会随时被内核唤醒执行，也会像中断一样打断正在执行的代码 |
| 内核抢占 | 内核具有抢占性，发生抢占时，如果抢占的线程和被抢占的线程在相同的临界区，就产生了竞争条件 |
| 睡眠及用户空间的同步 | 用户进程睡眠后，调度程序会唤醒一个新的用户进程，新的用户进程和睡眠的进程可能在同一个临界区中 |
| 对称多处理 | 2个或多个处理器可以同时执行相同的代码 |

为了在编写内核代码时避免出现竞争条件，在**编写代码之前就要考虑好临界区在哪，以及怎么加锁**。

在编写完代码后再加锁是非常困难的，很可能还会导致部分代码重写。

编写内核代码时，时时记着下面这些问题：

1. 这个数据是不是全局的？除了当前线程以外，其他线程能不能访问它？
2. 这个数据会不会在进程上下文或者中断上下文中共享？它是不是要在两个不同的中断处理程序中共享？
3. 进程在访问数据时可不可能被抢占？被调度的新程序会不会访问同一数据？
4. 当前进程会不会睡眠(或者阻塞)在某些资源上，如果是，它会让共享数据处于何种状态？
5. 怎样防止数据失控？
6. 如果这个函数又在另一个处理器上被调度将会发生什么？

### 3. 死锁

死锁就是所有线程都在相互等待释放资源，导致谁也无法继续执行下去。

下面一些简单的规则可以帮助我们避免死锁：

1. 如果有多个锁的话，尽量确保每个线程都是按相同的顺序加锁，按加锁相反的顺序解锁。(即加锁a->b->c，解锁c->b->a)
2. 防止发生饥饿。即设置一个超时时间，防止一直等待下去。
3. 不要重复请求同一个锁。
4. 设计应力求简单。加锁的方案越复杂就越容易出现死锁。

### 4. 锁的粒度

在加锁的时候，不仅要避免死锁，还需要考虑加锁的粒度。

锁的粒度对系统的可扩展性有很大影响，在加锁的时候，要考虑一下这个锁是否会被多个线程频繁的争用。

如果锁有可能会被频繁争用，就需要将锁的粒度细化。

细化后的锁在多处理器的情况下，性能会有所提升。

举个例子说明一下：比如给一个链表加锁，同时有A，B，C 3个线程频繁访问这个链表。

那么当A，B，C 3个线程同时访问这个链表时，如果A获得了锁，那么B，C线程只能等待A释放了锁后才能访问这个链表。

如果A，B，C 3个线程访问的是这个链表的不同节点（比如A是修改节点listA，B是删除节点listB，C是追加节点listC），

并且这3个节点不是连续的，那么3个线程同时运行是不会有问题的。

这种情况下就可以细化这个锁，把加在链表上的锁去掉，改成把锁加在链表的每个节点上。（也就是锁粒度的细化）

那么，上述的情况下，A，B，C 3个线程就可以同时访问各自的节点，特别是在多处理器的情况下，性能会有显著提高。

最后还有一点需要提醒的是，锁的粒度越细，系统开销越大，程序也越复杂，所以对于争用不是很频繁的锁，就没有必要细化了。

# [（十）- 内核同步方法](http://www.cnblogs.com/wang_yb/archive/2013/05/01/3052865.html)

内核中提供了多种方法来防止竞争条件，理解了这些方法的使用场景有助于我们在编写内核代码时选用合适的同步方法，

从而即可保证代码中临界区的安全，同时也让性能的损失降到最低。

**主要内容：**

* 原子操作
* 自旋锁
* 读写自旋锁
* 信号量
* 读写信号量
* 互斥体
* 完成变量
* 大内核锁
* 顺序锁
* 禁止抢占
* 顺序和屏障
* 总结

### 1. 原子操作

原子操作是由编译器来保证的，保证一个线程对数据的操作不会被其他线程打断。

原子操作有2类：

1. 原子整数操作，有32位和64位。头文件分别为<asm/atomic.h>和<asm/atomic64.h>
2. 原子位操作。头文件 <asm/bitops.h>

原子操作的api很简单，参见相应的头文件即可。

原子操作头文件与具体的体系结构有关，比如x86架构的相关头文件在 arch/x86/include/asm/\*.h

### 2. 自旋锁

原子操作只能用于临界区只有一个变量的情况，实际应用中，临界区的情况要复杂的多。

对于复杂的临界区，linux内核中也提供了多种同步方法，自旋锁就是其中一种。

自旋锁的特点就是当一个线程获取了锁之后，其他试图获取这个锁的线程一直在循环等待获取这个锁，直至锁重新可用。

由于线程实在一直循环的获取这个锁，所以会造成CPU处理时间的浪费，因此最好将自旋锁用于能很快处理完的临界区。

自旋锁的实现与体系结构有关，所以相应的头文件 <asm/spinlock.h> 位于相关体系结构的代码中。

自旋锁使用时有2点需要注意：

1. 自旋锁是不可递归的，递归的请求同一个自旋锁会自己锁死自己。
2. 线程获取自旋锁之前，要禁止当前处理器上的中断。（防止获取锁的线程和中断形成竞争条件）   
     比如：当前线程获取自旋锁后，在临界区中被中断处理程序打断，中断处理程序正好也要获取这个锁，   
     于是中断处理程序会等待当前线程释放锁，而当前线程也在等待中断执行完后再执行临界区和释放锁的代码。

中断处理下半部的操作中使用自旋锁尤其需要小心：

1. 下半部处理和进程上下文共享数据时，由于下半部的处理可以抢占进程上下文的代码，   
     所以进程上下文在对共享数据加锁前要禁止下半部的执行，解锁时再允许下半部的执行。
2. 中断处理程序（上半部）和下半部处理共享数据时，由于中断处理（上半部）可以抢占下半部的执行，   
     所以下半部在对共享数据加锁前要禁止中断处理（上半部），解锁时再允许中断的执行。
3. 同一种tasklet不能同时运行，所以同类tasklet中的共享数据不需要保护。
4. 不同类tasklet中共享数据时，其中一个tasklet获得锁后，不用禁止其他tasklet的执行，因为同一个处理器上不会有tasklet相互抢占的情况
5. 同类型或者非同类型的软中断在共享数据时，也不用禁止下半部，因为同一个处理器上不会有软中断互相抢占的情况

自旋锁方法列表如下：

|  |  |
| --- | --- |
| **方法** | **描述** |
| spin\_lock() | 获取指定的自旋锁 |
| spin\_lock\_irq() | 禁止本地中断并获取指定的锁 |
| spin\_lock\_irqsave() | 保存本地中断的当前状态，禁止本地中断，并获取指定的锁 |
| spin\_unlock() | 释放指定的锁 |
| spin\_unlock\_irq() | 释放指定的锁，并激活本地中断 |
| spin\_unlock\_irqstore() | 释放指定的锁，并让本地中断恢复到以前状态 |
| spin\_lock\_init() | 动态初始化指定的spinlock\_t |
| spin\_trylock() | 试图获取指定的锁，如果未获取，则返回0 |
| spin\_is\_locked() | 如果指定的锁当前正在被获取，则返回非0，否则返回0 |

### 3. 读写自旋锁

1. 读写自旋锁除了和普通自旋锁一样有自旋特性以外，还有以下特点：   
   读锁之间是共享的   
     即一个线程持有了读锁之后，其他线程也可以以读的方式持有这个锁
2. 写锁之间是互斥的   
     即一个线程持有了写锁之后，其他线程不能以读或者写的方式持有这个锁
3. 读写锁之间是互斥的   
     即一个线程持有了读锁之后，其他线程不能以写的方式持有这个锁

**注**：读写锁要分别使用，不能混合使用，否则会造成死锁。

正常的使用方法：

[复制代码](javascript:void(0);)

DEFINE\_RWLOCK(mr\_rwlock);

read\_lock(&mr\_rwlock);

/\* 临界区(只读).... \*/

read\_unlock(&mr\_rwlock);

write\_lock(&mr\_lock);

/\* 临界区(读写)... \*/

write\_unlock(&mr\_lock);

[复制代码](javascript:void(0);)

混合使用时：

[复制代码](javascript:void(0);)

/\* 获取一个读锁 \*/

read\_lock(&mr\_lock);

/\* 在获取写锁的时候，由于读写锁之间是互斥的，

\* 所以写锁会一直自旋等待读锁的释放，

\* 而此时读锁也在等待写锁获取完成后继续下面的代码。

\* 因此造成了读写锁的互相等待，形成了死锁。

\*/

write\_lock(&mr\_lock);

[复制代码](javascript:void(0);)

读写锁相关文件参照 各个体系结构中的 <asm/rwlock.h>

读写锁的相关函数如下：

|  |  |
| --- | --- |
| **方法** | **描述** |
| read\_lock() | 获取指定的读锁 |
| read\_lock\_irq() | 禁止本地中断并获得指定读锁 |
| read\_lock\_irqsave() | 存储本地中断的当前状态，禁止本地中断并获得指定读锁 |
| read\_unlock() | 释放指定的读锁 |
| read\_unlock\_irq() | 释放指定的读锁并激活本地中断 |
| read\_unlock\_irqrestore() | 释放指定的读锁并将本地中断恢复到指定前的状态 |
| write\_lock() | 获得指定的写锁 |
| write\_lock\_irq() | 禁止本地中断并获得指定写锁 |
| write\_lock\_irqsave() | 存储本地中断的当前状态，禁止本地中断并获得指定写锁 |
| write\_unlock() | 释放指定的写锁 |
| write\_unlock\_irq() | 释放指定的写锁并激活本地中断 |
| write\_unlock\_irqrestore() | 释放指定的写锁并将本地中断恢复到指定前的状态 |
| write\_trylock() | 试图获得指定的写锁；如果写锁不可用，返回非0值 |
| rwlock\_init() | 初始化指定的rwlock\_t |

### 4. 信号量

信号量也是一种锁，和自旋锁不同的是，线程获取不到信号量的时候，不会像自旋锁一样循环的去试图获取锁，

而是进入睡眠，直至有信号量释放出来时，才会唤醒睡眠的线程，进入临界区执行。

由于使用信号量时，线程会睡眠，所以等待的过程不会占用CPU时间。所以信号量适用于等待时间较长的临界区。

信号量消耗的CPU时间的地方在于使线程睡眠和唤醒线程，

如果 （使线程睡眠 + 唤醒线程）的CPU时间 > 线程自旋等待的CPU时间，那么可以考虑使用自旋锁。

信号量有二值信号量和计数信号量2种，其中二值信号量比较常用。

**二值信号量**表示信号量只有2个值，即0和1。信号量为1时，表示临界区可用，信号量为0时，表示临界区不可访问。

二值信号量表面看和自旋锁很相似，区别在于争用自旋锁的线程会一直循环尝试获取自旋锁，

而争用信号量的线程在信号量为0时，会进入睡眠，信号量可用时再被唤醒。

**计数信号量**有个计数值，比如计数值为5，表示同时可以有5个线程访问临界区。

信号量相关函数参照: <linux/semaphore.h> 实现方法参照：kernel/semaphore.c

使用信号量的方法如下：

[复制代码](javascript:void(0);)

/\* 定义并声明一个信号量，名字为mr\_sem，用于信号量计数 \*/

static DECLARE\_MUTEX(mr\_sem);

/\* 试图获取信号量....， 信号未获取成功时，进入睡眠

\* 此时，线程状态为 TASK\_INTERRUPTIBLE

\*/

down\_interruptible(&mr\_sem);

/\* 这里也可以用：

\* down(&mr\_sem);

\* 这个方法把线程状态置为 TASK\_UNINTERRUPTIBLE 后睡眠

\*/

/\* 临界区 ... \*/

/\* 释放给定的信号量 \*/

up(&mr\_sem);

[复制代码](javascript:void(0);)

一般用的比较多的是down\_interruptible()方法，因为以 TASK\_UNINTERRUPTIBLE 方式睡眠无法被信号唤醒。

对于 TASK\_INTERRUPTIBLE 和 TASK\_UNINTERRUPTIBLE 补充说明一下：

* TASK\_INTERRUPTIBLE - 可打断睡眠，可以接受信号并被唤醒，也可以在等待条件全部达成后被显式唤醒(比如wake\_up()函数)。
* TASK\_UNINTERRUPTIBLE - 不可打断睡眠，只能在等待条件全部达成后被显式唤醒(比如wake\_up()函数)。

信号量方法如下：

|  |  |
| --- | --- |
| **方法** | **描述** |
| sema\_init(struct semaphore \*, int) | 以指定的计数值初始化动态创建的信号量 |
| init\_MUTEX(struct semaphore \*) | 以计数值1初始化动态创建的信号量 |
| init\_MUTEX\_LOCKED(struct semaphore \*) | 以计数值0初始化动态创建的信号量（初始为加锁状态） |
| down\_interruptible(struct semaphore \*) | 以试图获得指定的信号量，如果信号量已被争用，则进入可中断睡眠状态 |
| down(struct semaphore \*) | 以试图获得指定的信号量，如果信号量已被争用，则进入不可中断睡眠状态 |
| down\_trylock(struct semaphore \*) | 以试图获得指定的信号量，如果信号量已被争用，则立即返回非0值 |
| up(struct semaphore \*) | 以释放指定的信号量，如果睡眠队列不空，则唤醒其中一个任务 |

信号量结构体具体如下：

/\* Please don't access any members of this structure directly \*/

struct semaphore {

spinlock\_t lock;

unsigned int count;

struct list\_head wait\_list;

};

可以发现信号量结构体中有个自旋锁，这个自旋锁的作用是保证信号量的down和up等操作不会被中断处理程序打断。

### 5. 读写信号量

读写信号量和信号量之间的关系 与 读写自旋锁和普通自旋锁之间的关系 差不多。

读写信号量都是二值信号量，即计数值最大为1，增加读者时，计数器不变，增加写者，计数器才减一。

也就是说读写信号量保护的临界区，最多只有一个写者，但可以有多个读者。

读写信号量的相关内容参见：<asm/rwsem.h> 具体实现与硬件体系结构有关。

### 6. 互斥体

互斥体也是一种可以睡眠的锁，相当于二值信号量，只是提供的API更加简单，使用的场景也更严格一些，如下所示：

1. mutex的计数值只能为1，也就是最多只允许一个线程访问临界区
2. 在同一个上下文中上锁和解锁
3. 不能递归的上锁和解锁
4. 持有个mutex时，进程不能退出
5. mutex不能在中断或者下半部中使用，也就是mutex只能在进程上下文中使用
6. mutex只能通过官方API来管理，不能自己写代码操作它

在面对**互斥体和信号量的选择**时，只要满足互斥体的使用场景就尽量优先使用互斥体。

在面对**互斥体和自旋锁的选择**时，参见下表：

|  |  |
| --- | --- |
| **需求** | **建议的加锁方法** |
| 低开销加锁 | 优先使用自旋锁 |
| 短期锁定 | 优先使用自旋锁 |
| 长期加锁 | 优先使用互斥体 |
| 中断上下文中加锁 | 使用自旋锁 |
| 持有锁需要睡眠 | 使用互斥体 |

互斥体头文件：<linux/mutex.h>

常用的互斥体方法如下：

|  |  |
| --- | --- |
| **方法** | **描述** |
| mutex\_lock(struct mutex \*) | 为指定的mutex上锁，如果锁不可用则睡眠 |
| mutex\_unlock(struct mutex \*) | 为指定的mutex解锁 |
| mutex\_trylock(struct mutex \*) | 试图获取指定的mutex，如果成功则返回1；否则锁被获取，返回0 |
| mutex\_is\_locked(struct mutex \*) | 如果锁已被争用，则返回1；否则返回0 |

### 7. 完成变量

完成变量的机制类似于信号量，

比如一个线程A进入临界区之后，另一个线程B会在完成变量上等待，线程A完成了任务出了临界区之后，使用完成变量来唤醒线程B。

完成变量的头文件：<linux/completion.h>

完成变量的API也很简单：

|  |  |
| --- | --- |
| **方法** | **描述** |
| init\_completion(struct completion \*) | 初始化指定的动态创建的完成变量 |
| wait\_for\_completion(struct completion \*) | 等待指定的完成变量接受信号 |
| complete(struct completion \*) | 发信号唤醒任何等待任务 |

使用完成变量的例子可以参考：kernel/sched.c 和 kernel/fork.c

一般在2个任务需要简单同步的情况下，可以考虑使用完成变量。

### 8. 大内核锁

大内核锁已经不再使用，只存在与一些遗留的代码中。

### 9. 顺序锁

顺序锁为读写共享数据提供了一种简单的实现机制。

之前提到的读写自旋锁和读写信号量，在读锁被获取之后，写锁是不能再被获取的，

也就是说，必须等所有的读锁释放后，才能对临界区进行写入操作。

顺序锁则与之不同，读锁被获取的情况下，写锁仍然可以被获取。

使用顺序锁的读操作在读之前和读之后都会检查顺序锁的序列值，如果前后值不符，则说明在读的过程中有写的操作发生，

那么读操作会重新执行一次，直至读前后的序列值是一样的。

do

{

/\* 读之前获取 顺序锁foo 的序列值 \*/

seq = read\_seqbegin(&foo);

...

} while(read\_seqretry(&foo, seq)); /\* 顺序锁foo此时的序列值!=seq 时返回true，反之返回false \*/

顺序锁优先保证写锁的可用，所以适用于那些读者很多，写者很少，且写优于读的场景。

顺序锁的使用例子可以参考：kernel/timer.c和kernel/time/tick-common.c文件

### 10. 禁止抢占

其实使用自旋锁已经可以防止内核抢占了，但是有时候仅仅需要禁止内核抢占，不需要像自旋锁那样连中断都屏蔽掉。

这时候就需要使用禁止内核抢占的方法了：

|  |  |
| --- | --- |
| **方法** | **描述** |
| preempt\_disable() | 增加抢占计数值，从而禁止内核抢占 |
| preempt\_enable() | 减少抢占计算，并当该值降为0时检查和执行被挂起的需调度的任务 |
| preempt\_enable\_no\_resched() | 激活内核抢占但不再检查任何被挂起的需调度的任务 |
| preempt\_count() | 返回抢占计数 |

这里的preempt\_disable()和preempt\_enable()是可以嵌套调用的，disable和enable的次数最终应该是一样的。

禁止抢占的头文件参见：<linux/preempt.h>

### 11. 顺序和屏障

对于一段代码，编译器或者处理器在编译和执行时可能会对执行顺序进行一些优化，从而使得代码的执行顺序和我们写的代码有些区别。

一般情况下，这没有什么问题，但是在并发条件下，可能会出现取得的值与预期不一致的情况

比如下面的代码：

[复制代码](javascript:void(0);)

/\*

\* 线程A和线程B共享的变量 a和b

\* 初始值 a=1, b=2

\*/

int a = 1, b = 2;

/\*

\* 假设线程A 中对 a和b的操作

\*/

void Thread\_A()

{

a = 5;

b = 4;

}

/\*

\* 假设线程B 中对 a和b的操作

\*/

void Thread\_B()

{

if (b == 4)

printf("a = %d\n", a);

}

[复制代码](javascript:void(0);)

由于编译器或者处理器的优化，线程A中的赋值顺序可能是b先赋值后，a才被赋值。

所以如果线程A中 b=4; 执行完，a=5; 还没有执行的时候，线程B开始执行，那么线程B打印的是a的初始值1。

这就与我们预期的不一致了，我们预期的是a在b之前赋值，所以线程B要么不打印内容，如果打印的话，a的值应该是5。

在某些并发情况下，为了保证代码的执行顺序，引入了一系列屏障方法来阻止编译器和处理器的优化。

|  |  |
| --- | --- |
| **方法** | **描述** |
| rmb() | 阻止跨越屏障的载入动作发生重排序 |
| read\_barrier\_depends() | 阻止跨越屏障的具有数据依赖关系的载入动作重排序 |
| wmb() | 阻止跨越屏障的存储动作发生重排序 |
| mb() | 阻止跨越屏障的载入和存储动作重新排序 |
| smp\_rmb() | 在SMP上提供rmb()功能，在UP上提供barrier()功能 |
| smp\_read\_barrier\_depends() | 在SMP上提供read\_barrier\_depends()功能，在UP上提供barrier()功能 |
| smp\_wmb() | 在SMP上提供wmb()功能，在UP上提供barrier()功能 |
| smp\_mb() | 在SMP上提供mb()功能，在UP上提供barrier()功能 |
| barrier() | 阻止编译器跨越屏障对载入或存储操作进行优化 |

为了使得上面的小例子能正确执行，用上表中的函数修改线程A的函数即可：

[复制代码](javascript:void(0);)

/\*

\* 假设线程A 中对 a和b的操作

\*/

void Thread\_A()

{

a = 5;

mb();

/\*

\* mb()保证在对b进行载入和存储值(值就是4)的操作之前

\* mb()代码之前的所有载入和存储值的操作全部完成(即 a = 5;已经完成)

\* 只要保证a的赋值在b的赋值之前进行，那么线程B的执行结果就和预期一样了

\*/

b = 4;

}

[复制代码](javascript:void(0);)

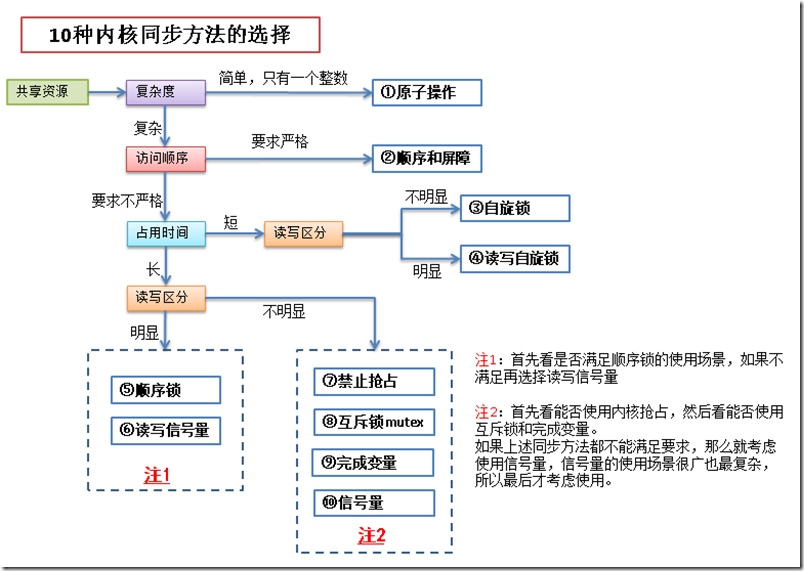
### 12. 总结

本节讨论了大约11种内核同步方法，除了大内核锁已经不再推荐使用之外，其他各种锁都有其适用的场景。

了解了各种同步方法的适用场景，才能正确的使用它们，使我们的代码在安全的保障下达到最优的性能。

同步的目的就是为了保障数据的安全，其实就是保障各个线程之间共享资源的安全，下面根据共享资源的情况来讨论一下10种同步方法的选择。

10种同步方法在图中分别用蓝色框标出。

[](https://images0.cnblogs.com/blog/83005/201305/01111554-c232cb4333a446fca92bbe6c03077863.png)

# [定时器和时间管理](http://www.cnblogs.com/wang_yb/archive/2013/05/10/3070373.html)

系统中有很多与时间相关的程序（比如定期执行的任务，某一时间执行的任务，推迟一段时间执行的任务），因此，时间的管理对于linux来说非常重要。

**主要内容：**

* 系统时间
* 定时器
* 定时器相关概念
* 定时器执行流程
* 实现程序延迟的方法
* 定时器和延迟的例子

### 1. 系统时间

系统中管理的时间有2种：实际时间和定时器。

#### 1.1  实际时间

实际时间就是现实中钟表上显示的时间，其实内核中并不常用这个时间，主要是用户空间的程序有时需要获取当前时间，

所以内核中也管理着这个时间。

实际时间的获取是在开机后，内核初始化时从**RTC**读取的。

内核读取这个时间后就将其放入内核中的 **xtime** 变量中，并且在系统的运行中不断更新这个值。

**注：**RTC就是实时时钟的缩写，它是用来存放系统时间的设备。一般和BIOS一样，由主板上的电池供电的，所以即使关机也可将时间保存。

实际时间存放的变量 **xtime** 在文件 kernel/time/timekeeping.c中。

[复制代码](javascript:void(0);)

/\* 按照16位对齐，其实就是2个long型的数据 \*/

struct timespec xtime \_\_attribute\_\_ ((aligned (16)));

/\* timespec结构体的定义如下， 参考 <linux/time.h> \*/

struct timespec {

\_\_kernel\_time\_t tv\_sec; /\* seconds \*/

long tv\_nsec; /\* nanoseconds \*/

};

/\* \_kernel\_time\_t 定义如下 \*/

typedef long \_\_kernel\_time\_t;

[复制代码](javascript:void(0);)

系统读写 **xtime** 时用的就是顺序锁。

[复制代码](javascript:void(0);)

/\* 写入 xtime 参考 do\_sometimeofday 方法 \*/

int do\_settimeofday(struct timespec \*tv)

{

/\* 省略 。。。。 \*/

write\_seqlock\_irqsave(&xtime\_lock, flags); /\* 获取写锁 \*/

/\* 更新 xtime \*/

write\_sequnlock\_irqrestore(&xtime\_lock, flags); /\* 释放写锁 \*/

/\* 省略 。。。。 \*/

return 0;

}

/\* 读取 xtime 参考 do\_gettimeofday 方法 \*/

void do\_gettimeofday(struct timeval \*tv)

{

struct timespec now;

getnstimeofday(&now); /\* 就是在这个方法中获取读锁，并读取 xtime \*/

tv->tv\_sec = now.tv\_sec;

tv->tv\_usec = now.tv\_nsec/1000;

}

void getnstimeofday(struct timespec \*ts)

{

/\* 省略 。。。。 \*/

/\* 顺序锁中读锁来循环获取 xtime，直至读取过程中 xtime 没有被改变过 \*/

do {

seq = read\_seqbegin(&xtime\_lock);

\*ts = xtime;

nsecs = timekeeping\_get\_ns();

/\* If arch requires, add in gettimeoffset() \*/

nsecs += arch\_gettimeoffset();

} while (read\_seqretry(&xtime\_lock, seq));

/\* 省略 。。。。 \*/

}

[复制代码](javascript:void(0);)

上述场景中，写锁必须要优先于读锁(因为 **xtime** 必须及时更新)，而且写锁的使用者很少(一般只有系统定期更新**xtime**的线程需要持有这个锁)。

这正是 顺序锁的应用场景。

#### 1.2 定时器

定时器是内核中主要使用的时间管理方法，通过定时器，可以有效的调度程序的执行。

动态定时器是内核中使用比较多的定时器，下面重点讨论的也是动态定时器。

### 2. 定时器

内核中的定时器有2种，静态定时器和动态定时器。

静态定时器一般执行了一些周期性的固定工作：

* 更新系统运行时间
* 更新实际时间
* 在SMP系统上，平衡各个处理器上的运行队列
* 检查当前进程是否用尽了自己的时间片，如果用尽，需要重新调度。
* 更新资源消耗和处理器时间统计值

动态定时器顾名思义，是在需要时（一般是推迟程序执行）动态创建的定时器，使用后销毁（一般都是只用一次）。

一般我们在内核代码中使用的定时器基本都是动态定时器，下面重点讨论动态定时器相关的概念和使用方法。

### 3. 定时器相关概念

定时器的使用中，下面3个概念非常重要：

1. HZ
2. jiffies
3. 时间中断处理程序

#### 3.1 HZ

节拍率(HZ)是时钟中断的频率，表示的一秒内时钟中断的次数。

比如 HZ=100 表示一秒内触发100次时钟中断程序。

HZ的值一般与体系结构有关，x86 体系结构一般定义为 100，参考文件 include/asm-generic/param.h

HZ值的大小的设置过程其实就是平衡 精度和性能 的过程，并不是HZ值越高越好。

|  |  |  |
| --- | --- | --- |
| **HZ值** | **优势** | **劣势** |
| 高HZ | 时钟中断程序运行的更加频繁，依赖时间执行的程序更加精确，  对资源消耗和系统运行时间的统计更加精确。 | 时钟中断执行的频繁，增加系统负担  时钟中断占用的CPU时间过多 |

此外，有一点需要注意，内核中使用的HZ可能和用户空间中定义的HZ值不一致，为了避免用户空间取得错误的时间，

内核中也定义了 USER\_HZ，即用户空间使用的HZ值。

一般来说，USER\_HZ 和 HZ 都是相差整数倍，内核中通过函数 **jiffies\_to\_clock\_t** 来将内核来将内核中的 jiffies转为 用户空间 jiffies

[复制代码](javascript:void(0);)

/\* 参见文件： kernel/time.c \*

//\*

\* Convert jiffies/jiffies\_64 to clock\_t and back.

\*/

clock\_t jiffies\_to\_clock\_t(unsigned long x)

{

#if (TICK\_NSEC % (NSEC\_PER\_SEC / USER\_HZ)) == 0

# if HZ < USER\_HZ

return x \* (USER\_HZ / HZ);

# else

return x / (HZ / USER\_HZ);

# endif

#else

return div\_u64((u64)x \* TICK\_NSEC, NSEC\_PER\_SEC / USER\_HZ);

#endif

}

EXPORT\_SYMBOL(jiffies\_to\_clock\_t);

[复制代码](javascript:void(0);)

#### 3.2 jiffies

jiffies用来记录自系统启动以来产生的总节拍数。比如系统启动了 N 秒，那么 jiffies就为 N×HZ

jiffies的相关定义参考头文件 **<linux/jiffies.h>**  include/linux/jiffies.h

/\* 64bit和32bit的jiffies定义如下 \*/

extern u64 \_\_jiffy\_data jiffies\_64;

extern unsigned long volatile \_\_jiffy\_data jiffies;

使用定时器时一般都是以jiffies为单位来延迟程序执行的，比如延迟5个节拍后执行的话，执行时间就是 jiffies+5

32位的jiffies的最大值为 2^32-1，在使用时有可能会出现回绕的问题。

比如下面的代码：

[复制代码](javascript:void(0);)

unsigned long timeout = jiffies + HZ/2; /\* 设置超时时间为 0.5秒 \*/

while (timeout < jiffies)

{

/\* 还没有超时，继续执行任务 \*/

}

/\* 执行超时后的任务 \*/

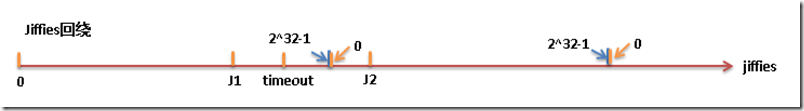
[复制代码](javascript:void(0);)

正常情况下，上面的代码没有问题。当jiffies接近最大值的时候，就会出现回绕问题。

由于是**unsinged long**类型，所以jiffies达到最大值后会变成0然后再逐渐变大，如下图所示：

[](https://images0.cnblogs.com/blog/83005/201305/10075541-95313de661534d89974b4467486ed5f2.png)

所以在上述的循环代码中，会出现如下情况：

[](https://images0.cnblogs.com/blog/83005/201305/10075542-8125a5242642415e8d91258926deaa6e.png)

1. 循环中第一次比较时，jiffies = J1，没有超时
2. 循环中第二次比较时，jiffies = J2，实际已经超时了，但是由于jiffies超过的最大值后又从0开始，所以J2远远小于timeout
3. while循环会执行很长时间(> 2^32-1 个节拍)不会结束，几乎相当于死循环了

为了回避回扰的问题，可以使用**<linux/jiffies.h>**头文件中提供的 **time\_after**，**time\_before**等宏

[复制代码](javascript:void(0);)

#define time\_after(a,b) \

(typecheck(unsigned long, a) && \

typecheck(unsigned long, b) && \

((long)(b) - (long)(a) < 0))

#define time\_before(a,b) time\_after(b,a)

#define time\_after\_eq(a,b) \

(typecheck(unsigned long, a) && \

typecheck(unsigned long, b) && \

((long)(a) - (long)(b) >= 0))

#define time\_before\_eq(a,b) time\_after\_eq(b,a)

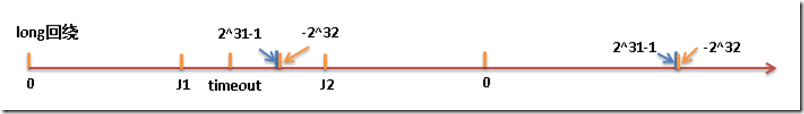
[复制代码](javascript:void(0);)

上述代码的原理其实就是将 **unsigned long** 类型转换为 **long** 类型来避免回扰带来的错误，

**long** 类型超过最大值时变化趋势如下：

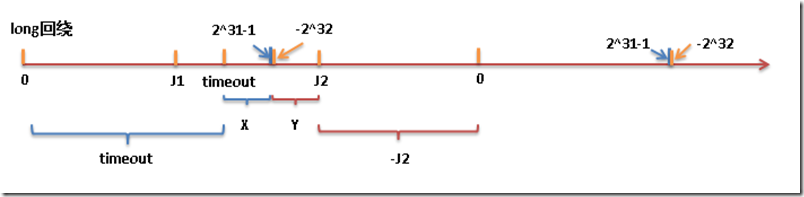
[](https://images0.cnblogs.com/blog/83005/201305/10075542-e6e974f29a8b46a3a202b31acf2c9056.png)

**long** 型的数据的回绕会出现在 2^31-1 变为 -2^32 的时候，如下图所示：

[](https://images0.cnblogs.com/blog/83005/201305/10075543-dc3a73ecb64e40e29299d6eba8d04a11.png)

1. 第一次比较时，jiffies = J1，没有超时
2. 第二次比较时，jiffies = J2，一般 J2 是负数   
   理论上 (long)timeout - (long)J2 = 正数 - 负数 = 正数（result）   
   但是，这个正数（result）一般会大于 2^31 - 1，所以long型的result又发生了一次回绕，变成了负数。   
   除非timeout和J2之间的间隔 > 2^32 个节拍，result的值才会为正数(注1)。

**注1**：result的值为正数时，必须是在result的值 小于 2^31-1 的情况下，大于 2^31-1 会发生回绕。

[](https://images0.cnblogs.com/blog/83005/201305/10075543-4879a18c9b6c4addbe4f260a4bace383.png)

上图中 X + Y 表示timeout 和 J2之间经过的节拍数。

result 小于 2^31-1 ，也就是 timeout - J2 < 2^31 – 1

timeout 和 -J2 表示的节拍数如上图所示。(因为J2是负数，所有-J2表示上图所示范围的值)

因为 timeout + X + Y - J2 = 2^31-1 + 2^32

所以 timeout - J2 < 2^31 - 1 时， X + Y > 2^32

也就是说，当timeout和J2之间经过至少 2^32 个节拍后，result才可能变为正数。

timeout和J2之间相差这么多节拍是不可能的(不信可以用HZ将这些节拍换算成秒就知道了。。。)

利用time\_after宏就可以巧妙的避免回绕带来的超时判断问题，将之前的代码改成如下代码即可：

[复制代码](javascript:void(0);)

unsigned long timeout = jiffies + HZ/2; /\* 设置超时时间为 0.5秒 \*/

while (time\_after(jiffies, timeout))

{

/\* 还没有超时，继续执行任务 \*/

}

/\* 执行超时后的任务 \*/

[复制代码](javascript:void(0);)

#### 3.3 时钟中断处理程序

时钟中断处理程序作为系统定时器而注册到内核中，体系结构的不同，可能时钟中断处理程序中处理的内容不同。

但是以下这些基本的工作都会执行：

* 获得 xtime\_lock 锁，以便对访问 jiffies\_64 和墙上时间 xtime 进行保护
* 需要时应答或重新设置系统时钟
* 周期性的使用墙上时间更新实时时钟
* 调用 tick\_periodic()

tick\_periodic函数位于： kernel/time/tick-common.c 中

[复制代码](javascript:void(0);)

static void tick\_periodic(int cpu)

{

if (tick\_do\_timer\_cpu == cpu) {

write\_seqlock(&xtime\_lock);

/\* Keep track of the next tick event \*/

tick\_next\_period = ktime\_add(tick\_next\_period, tick\_period);

do\_timer(1);

write\_sequnlock(&xtime\_lock);

}

update\_process\_times(user\_mode(get\_irq\_regs()));

profile\_tick(CPU\_PROFILING);

}

[复制代码](javascript:void(0);)

其中最重要的是 do\_timer 和 update\_process\_times 函数。

我了解的步骤进行了简单的注释。

[复制代码](javascript:void(0);)

void do\_timer(unsigned long ticks)

{

/\* jiffies\_64 增加指定ticks \*/

jiffies\_64 += ticks;

/\* 更新实际时间 \*/

update\_wall\_time();

/\* 更新系统的平均负载值 \*/

calc\_global\_load();

}

void update\_process\_times(int user\_tick)

{

struct task\_struct \*p = current;

int cpu = smp\_processor\_id();

/\* 更新当前进程占用CPU的时间 \*/

account\_process\_tick(p, user\_tick);

/\* 同时触发软中断，处理所有到期的定时器 \*/

run\_local\_timers();

rcu\_check\_callbacks(cpu, user\_tick);

printk\_tick();

/\* 减少当前进程的时间片数 \*/

scheduler\_tick();

run\_posix\_cpu\_timers(p);

}

[复制代码](javascript:void(0);)

### 4. 定时器执行流程

这里讨论的定时器执行流程是动态定时器的执行流程。

#### 4.1 定时器的定义

定时器在内核中用一个链表来保存的，链表的每个节点都是一个定时器。

参见头文件 **<linux/timer.h>**

[复制代码](javascript:void(0);)

struct timer\_list {

struct list\_head entry;

unsigned long expires;

void (\*function)(unsigned long);

unsigned long data;

struct tvec\_base \*base;

#ifdef CONFIG\_TIMER\_STATS

void \*start\_site;

char start\_comm[16];

int start\_pid;

#endif

#ifdef CONFIG\_LOCKDEP

struct lockdep\_map lockdep\_map;

#endif

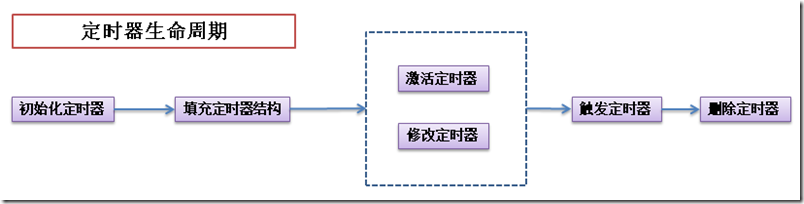
};

[复制代码](javascript:void(0);)

通过加入条件编译的参数，可以追加一些调试信息。

#### 4.2 定时器的生命周期

一个动态定时器的生命周期中，一般会经过下面的几个步骤：

[](https://images0.cnblogs.com/blog/83005/201305/10075544-278a7b346677402a861343e0fdce0314.png)

1. 初始化定时器：

struct timer\_list my\_timer; /\* 定义定时器 \*/

init\_timer(&my\_timer); /\* 初始化定时器 \*/

2. 填充定时器：

my\_timer.expires = jiffies + delay; /\* 定义超时的节拍数 \*/

my\_timer.data = 0; /\* 给定时器函数传入的参数 \*/

my\_timer.function = my\_function; /\* 定时器超时时，执行的自定义函数 \*/

/\* 从定时器结构体中，我们可以看出这个函数的原型应该如下所示： \*/

void my\_function(unsigned long data);

3. 激活定时器和修改定时器：

激活定时器之后才会被触发，否则定时器不会执行。

修改定时器主要是修改定时器的延迟时间，修改定时器后，不管原先定时器有没有被激活，都会处于激活状态。

填充定时器结构之后，可以只激活定时器，也可以只修改定时器，也可以激活定时器后再修改定时器。

所以填充定时器结构和触发定时器之间的步骤，也就是虚线框中的步骤是不确定的。

add\_timer(&my\_timer); /\* 激活定时器 \*/

mod\_timer(&my\_timer, jiffies + new\_delay); /\* 修改定时器，设置新的延迟时间 \*/

4. 触发定时器：

每次时钟中断处理程序会检查已经激活的定时器是否超时，如果超时就执行定时器结构中的自定义函数。

5. 删除定时器：

激活和未被激活的定时器都可以被删除，已经超时的定时器会自动删除，不用特意去删除。

/\*

\* 删除激活的定时器时，此函数返回1

\* 删除未激活的定时器时，此函数返回0

\*/

del\_timer(&my\_timer);

在多核处理器上用 del\_timer 函数删除定时器时，可能在删除时正好另一个CPU核上的时钟中断处理程序正在执行这个定时器，于是就形成了竞争条件。

为了避免竞争条件，建议使用 del\_timer\_sync 函数来删除定时器。

del\_timer\_sync 函数会等待其他处理器上的定时器处理程序全部结束后，才删除指定的定时器。

/\*

\* 和del\_timer 不同，del\_timer\_sync 不能在中断上下文中执行

\*/

del\_timer\_sync(&my\_timer);

### 5. 实现程序延迟的方法

内核中有个利用定时器实现延迟的函数 schedule\_timeout

这个函数会将当前的任务睡眠到指定时间后唤醒，所以等待时不会占用CPU时间。

/\* 将任务设置为可中断睡眠状态 \*/

set\_current\_state(TASK\_INTERRUPTIBLE);

/\* 小睡一会儿，“s“秒后唤醒 \*/

schedule\_timeout(s\*HZ);

查看 schedule\_timeout 函数的实现方法，可以看出是如何使用定时器的。

[复制代码](javascript:void(0);)

signed long \_\_sched schedule\_timeout(signed long timeout)

{

/\* 定义一个定时器 \*/

struct timer\_list timer;

unsigned long expire;

switch (timeout)

{

case MAX\_SCHEDULE\_TIMEOUT:

/\*

\* These two special cases are useful to be comfortable

\* in the caller. Nothing more. We could take

\* MAX\_SCHEDULE\_TIMEOUT from one of the negative value

\* but I' d like to return a valid offset (>=0) to allow

\* the caller to do everything it want with the retval.

\*/

schedule();

goto out;

default:

/\*

\* Another bit of PARANOID. Note that the retval will be

\* 0 since no piece of kernel is supposed to do a check

\* for a negative retval of schedule\_timeout() (since it

\* should never happens anyway). You just have the printk()

\* that will tell you if something is gone wrong and where.

\*/

if (timeout < 0) {

printk(KERN\_ERR "schedule\_timeout: wrong timeout "

"value %lx\n", timeout);

dump\_stack();

current->state = TASK\_RUNNING;

goto out;

}

}

/\* 设置超时时间 \*/

expire = timeout + jiffies;

/\* 初始化定时器，超时处理函数是 process\_timeout，后面再补充说明一下这个函数 \*/

setup\_timer\_on\_stack(&timer, process\_timeout, (unsigned long)current);

/\* 修改定时器，同时会激活定时器 \*/

\_\_mod\_timer(&timer, expire, false, TIMER\_NOT\_PINNED);

/\* 将本任务睡眠，调度其他任务 \*/

schedule();

/\* 删除定时器，其实就是 del\_timer\_sync 的宏

del\_singleshot\_timer\_sync(&timer);

/\* Remove the timer from the object tracker \*/

destroy\_timer\_on\_stack(&timer);

timeout = expire - jiffies;

out:

return timeout < 0 ? 0 : timeout;

}

EXPORT\_SYMBOL(schedule\_timeout);

/\*

\* 超时处理函数 process\_timeout 里面只有一步操作，唤醒当前任务。

\* process\_timeout 的参数其实就是 当前任务的地址

\*/

static void process\_timeout(unsigned long \_\_data)

{

wake\_up\_process((struct task\_struct \*)\_\_data);

}

[复制代码](javascript:void(0);)

schedule\_timeout 一般用于延迟时间较长的程序。

这里的延迟时间较长是对于计算机而言的，其实也就是延迟大于 1 个节拍(jiffies)。

对于某些极其短暂的延迟，比如只有1ms，甚至1us，1ns的延迟，必须使用特殊的延迟方法。

1s = 1000ms = 1000000us = 1000000000ns (1秒=1000毫秒=1000000微秒=1000000000纳秒)

假设 HZ=100，那么 1个节拍的时间间隔是 1/100秒，大概10ms左右。

所以对于那些极其短暂的延迟，schedule\_timeout 函数是无法使用的。

好在内核对于这些短暂，精确的延迟要求也提供了相应的宏。

/\* 具体实现参见 include/linux/delay.h

\* 以及 arch/x86/include/asm/delay.h

\*/

#define mdelay(n) ...

#define udelay(n) ...

#define ndelay(n) ...

通过这些宏，可以简单的实现延迟，比如延迟 5ns，只需 ndelay(5); 即可。

这些短延迟的实现原理并不复杂，

首先，内核在启动时就计算出了当前处理器1秒能执行多少次循环，即 loops\_per\_jiffy

(loops\_per\_jiffy 的计算方法参见 init/main.c 文件中的 calibrate\_delay 方法)。

然后算出延迟 5ns 需要循环多少次，执行那么多次空循环即可达到延迟的效果。

loops\_per\_jiffy 的值可以在启动信息中看到：

[root@vbox ~]# dmesg | grep delay

Calibrating delay loop (skipped), value calculated using timer frequency.. 6387.58 BogoMIPS (lpj=3193792)

我的虚拟机中看到 (lpj=3193792)

### 6. 定时器和延迟的例子

下面的例子测试了短延迟，自定义定时器以及 schedule\_timeout 的使用：

[复制代码](javascript:void(0);)

#include <linux/sched.h>

#include <linux/timer.h>

#include <linux/jiffies.h>

#include <asm/param.h>

#include <linux/delay.h>

#include "kn\_common.h"

MODULE\_LICENSE("Dual BSD/GPL");

static void test\_short\_delay(void);

static void test\_delay(void);

static void test\_schedule\_timeout(void);

static void my\_delay\_function(unsigned long);

static int testdelay\_init(void)

{

printk(KERN\_ALERT "HZ in current system: %dHz\n", HZ);

/\* test short delay \*/

test\_short\_delay();

/\* test delay \*/

test\_delay();

/\* test schedule timeout \*/

test\_schedule\_timeout();

return 0;

}

static void testdelay\_exit(void)

{

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

print\_current\_time(0);

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

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

}

static void test\_short\_delay()

{

printk(KERN\_ALERT "jiffies [b e f o r e] short delay: %lu", jiffies);

ndelay(5);

printk(KERN\_ALERT "jiffies [a f t e r] short delay: %lu", jiffies);

}

static void test\_delay()

{

/\* 初始化定时器 \*/

struct timer\_list my\_timer;

init\_timer(&my\_timer);

/\* 填充定时器 \*/

my\_timer.expires = jiffies + 1\*HZ; /\* 2秒后超时函数执行 \*/

my\_timer.data = jiffies;

my\_timer.function = my\_delay\_function;

/\* 激活定时器 \*/

add\_timer(&my\_timer);

}

static void my\_delay\_function(unsigned long data)

{

printk(KERN\_ALERT "This is my delay function start......\n");

printk(KERN\_ALERT "The jiffies when init timer: %lu\n", data);

printk(KERN\_ALERT "The jiffies when timer is running: %lu\n", jiffies);

printk(KERN\_ALERT "This is my delay function end........\n");

}

static void test\_schedule\_timeout()

{

printk(KERN\_ALERT "This sample start at : %lu", jiffies);

/\* 睡眠2秒 \*/

set\_current\_state(TASK\_INTERRUPTIBLE);

printk(KERN\_ALERT "sleep 2s ....\n");

schedule\_timeout(2\*HZ);

printk(KERN\_ALERT "This sample end at : %lu", jiffies);

}

module\_init(testdelay\_init);

module\_exit(testdelay\_exit);

[复制代码](javascript:void(0);)

其中用到的 kn\_common.h 和 kn\_common.c 参见之前的博客 [《Linux内核设计与实现》读书笔记（六）- 内核数据结构](http://www.cnblogs.com/wang_yb/archive/2013/04/16/3023892.html)

Makefile如下：

[复制代码](javascript:void(0);)

# must complile on customize kernel

obj-m += mydelay.o

mydelay-objs := testdelay.o kn\_common.o

#generate the path

CURRENT\_PATH:=$(shell pwd)

#the current kernel version number

LINUX\_KERNEL:=$(shell uname -r)

#the absolute path

LINUX\_KERNEL\_PATH:=/usr/src/kernels/$(LINUX\_KERNEL)

#complie object

all:

make -C $(LINUX\_KERNEL\_PATH) M=$(CURRENT\_PATH) modules

rm -rf modules.order Module.symvers .\*.cmd \*.o \*.mod.c .tmp\_versions \*.unsigned

#clean

clean:

rm -rf modules.order Module.symvers .\*.cmd \*.o \*.mod.c \*.ko .tmp\_versions \*.unsigned

[复制代码](javascript:void(0);)

执行测试命令及查看结果的方法如下：(我的测试系统是 CentOS 6.3 x64)

[复制代码](javascript:void(0);)

[root@vbox chap11]# make

[root@vbox chap11]# insmod mydelay.ko

[root@vbox chap11]# rmmod mydelay.ko

[root@vbox chap11]# dmesg | tail -14

HZ in current system: 1000Hz

jiffies [b e f o r e] short delay: 4296079617

jiffies [a f t e r] short delay: 4296079617

This sample start at : 4296079619

sleep 2s ....

This is my delay function start......

The jiffies when init timer: 4296079619

The jiffies when timer is running: 4296080621

This is my delay function end........

This sample end at : 4296081622

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

2013-5-9 23:7:20

testdelay is exited!

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

[复制代码](javascript:void(0);)

结果说明：

1. 短延迟只延迟了 5ns，所以执行前后的jiffies是一样的。

jiffies [b e f o r e] short delay: 4296079617

jiffies [a f t e r] short delay: 4296079617

2. 自定义定时器延迟了1秒后执行自定义函数，由于我的系统 HZ=1000，所以jiffies应该相差1000

The jiffies when init timer: 4296079619

The jiffies when timer is running: 4296080621

实际上jiffies相差了 1002，多了2个节拍

3. schedule\_timeout 延迟了2秒，jiffies应该相差 2000

This sample start at : 4296079619

This sample end at : 4296081622

实际上jiffies相差了 2003，多了3个节拍

以上结果也说明了定时器的延迟并不是那么精确，差了2,3个节拍其实就是误差2,3毫秒(因为HZ=1000)

如果HZ=100的话，一个节拍是10毫秒，那么定时器的误差可能就发现不了了(误差只有2,3毫秒，没有超多1个节拍)。

# [内存管理](http://www.cnblogs.com/wang_yb/archive/2013/05/23/3095907.html)

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）。

所有内核的内存管理必须要简洁而且高效。

**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

[复制代码](javascript:void(0);)

/\*

\* 页中包含的成员非常多，还包含了一些联合体

\* 其中有些字段我暂时还不清楚含义，以后再补上。。。

\*/

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 \*/

union {

atomic\_t \_mapcount; /\* 已经映射到mms的pte的个数 \*/

struct { /\* 用于slab层 \*/

u16 inuse;

u16 objects;

};

};

union {

struct {

unsigned long private; /\* 此page作为私有数据时，指向私有数据 \*/

struct address\_space \*mapping; /\* 此page作为页缓存时，指向关联的address\_space \*/

};

#if USE\_SPLIT\_PTLOCKS

spinlock\_t ptl;

#endif

struct kmem\_cache \*slab; /\* 指向slab层 \*/

struct page \*first\_page; /\* 尾部复合页中的第一个页 \*/

};

union {

pgoff\_t index; /\* Our offset within mapping. \*/

void \*freelist; /\* SLUB: freelist req. slab lock \*/

};

struct list\_head lru; /\* 将页关联起来的链表项 \*/

#if defined(WANT\_PAGE\_VIRTUAL)

void \*virtual; /\* 页的虚拟地址 \*/

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

#ifdef CONFIG\_WANT\_PAGE\_DEBUG\_FLAGS

unsigned long debug\_flags; /\* Use atomic bitops on this \*/

#endif

#ifdef CONFIG\_KMEMCHECK

/\*

\* kmemcheck wants to track the status of each byte in a page; this

\* is a pointer to such a status block. NULL if not tracked.

\*/

void \*shadow;

#endif

};

[复制代码](javascript:void(0);)

物理内存的每个页都有一个对应的 page 结构，看似会在管理上浪费很多内存，其实细细算来并没有多少。

比如上面的page结构体，每个字段都算4个字节的话，总共40多个字节。(union结构只算一个字段)

那么对于一个页大小 4KB 的 4G内存来说，一个有 4\*1024\*1024 / 4 = 1048576 个page，

一个page 算40个字节，在管理内存上共消耗内存 40MB左右。

如果页的大小是 8KB 的话，消耗的内存只有 20MB 左右。相对于 4GB 来说并不算很多。

#### 1.2 区

页是内存管理的最小单元，但是并不是所有的页对于内核都一样。

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。

其实一般主要关注的区只有3个：

|  |  |  |
| --- | --- | --- |
| **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 ZONE\_HIGHMEM 区的内存，后面还会讨论。

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

[复制代码](javascript:void(0);)

/\*\*

\* @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)

#+end\_src

vmalloc的定义在 mm/vmalloc.c 中

#+begin\_src C

/\*\*

\* @size - 申请分配的字节数

\*/

void \*vmalloc(unsigned long size)

[复制代码](javascript:void(0);)

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。

因为kmalloc的物理地址和虚拟地址之间的映射比较简单，只需要将物理地址的第一页和虚拟地址的第一页关联起来即可。

而vmalloc由于物理地址是不连续的，所以要将物理地址的每一页都和虚拟地址关联起来才行。

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。

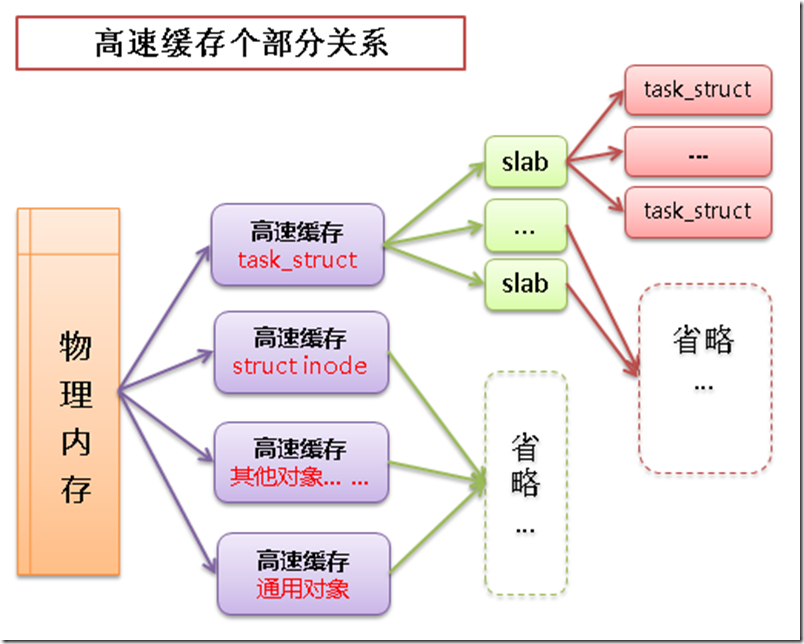
##### 2.3.1 slab层实现原理

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

[复制代码](javascript:void(0);)

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 \*/

};

[复制代码](javascript:void(0);)

slab层的应用主要有四个方法：

* 高速缓存的创建
* 从高速缓存中分配对象
* 向高速缓存释放对象
* 高速缓存的销毁

[复制代码](javascript:void(0);)

/\*\*

\* 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)

[复制代码](javascript:void(0);)

我做了创建高速缓存的例子，来尝试使用上面的几个函数。

测试代码如下：(其中用到的 kn\_common.h 和 kn\_common.c 参见之前的博客[《Linux内核设计与实现》读书笔记（六）- 内核数据结构](http://www.cnblogs.com/wang_yb/archive/2013/04/16/3023892.html" \t "_blank))

[复制代码](javascript:void(0);)

#include <linux/slab.h>

#include <linux/slab\_def.h>

#include "kn\_common.h"

MODULE\_LICENSE("Dual BSD/GPL");

#define MYSLAB "testslab"

static struct kmem\_cache \*myslab;

/\* 申请内存时调用的构造函数 \*/

static void ctor(void\* obj)

{

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

}

struct student

{

int id;

char\* name;

};

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

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");

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 \*/

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

print\_current\_time(0);

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);

[复制代码](javascript:void(0);)

Makefile文件如下：

[复制代码](javascript:void(0);)

# must complile on customize kernel

obj-m += myslab.o

myslab-objs := testslab.o kn\_common.o

#generate the path

CURRENT\_PATH:=$(shell pwd)

#the current kernel version number

LINUX\_KERNEL:=$(shell uname -r)

#the absolute path

LINUX\_KERNEL\_PATH:=/usr/src/kernels/$(LINUX\_KERNEL)

#complie object

all:

make -C $(LINUX\_KERNEL\_PATH) M=$(CURRENT\_PATH) modules

rm -rf modules.order Module.symvers .\*.cmd \*.o \*.mod.c .tmp\_versions \*.unsigned

#clean

clean:

rm -rf modules.order Module.symvers .\*.cmd \*.o \*.mod.c \*.ko .tmp\_versions \*.unsigned

[复制代码](javascript:void(0);)

执行测试代码：(我是在 centos6.3 x64 上实验的)

[复制代码](javascript:void(0);)

[root@vbox chap12]# make

[root@vbox chap12]# insmod myslab.ko

[root@vbox chap12]# dmesg | tail -220

# 可以看到第一次申请内存时，系统一次分配很多内存用于缓存(构造函数执行了多次)

[root@vbox chap12]# cat /proc/slabinfo | grep test #查看我们建立的缓存名在不在系统中

testslab 0 0 16 202 1 : tunables 120 60 0 : slabdata 0 0 0

[root@vbox chap12]# rmmod myslab.ko #卸载内核模块

[root@vbox chap12]# cat /proc/slabinfo | grep test #我们的缓存名已经不在系统中了

[复制代码](javascript:void(0);)

### 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。

[复制代码](javascript:void(0);)

/\* 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)

[复制代码](javascript:void(0);)

#### 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。

[复制代码](javascript:void(0);)

/\*\*

\* 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)

[复制代码](javascript:void(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

内核的内存分配和用户空间的内存分配相比有着更多的限制条件，同时也有着更高的性能要求。

下面讨论2个和用户空间不同的内存分配方式。

#### 4.1 内核栈上的静态分配

用户空间中一般不用担心栈上的内存不足，也不用担心内存的管理问题(比如内存越界之类的)，

即使出了异常也有内核来保证系统的正常运行。

而在内核空间则完全不一样，不仅栈空间有限，而且为了管理的效率和尽量减少问题的发生，

内核栈一般都是小而且固定的。

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，

配置为1页的好处是分配时比较简单，只有一页，不存在内存碎片的情况，因为一页是本就是分配的最小单位。

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。

假定有如下场景：

[复制代码](javascript:void(0);)

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 \*/

}

[复制代码](javascript:void(0);)

在上述场景下，可能会有以下的执行流程：

1. 刚开始 p == NULL
2. 线程A 执行到 [if (p == NULL)] ，刚进入 if 内的代码时被线程B 抢占   
     由于线程A 还没有执行 if 内的代码，所以 p 仍然是 NULL
3. 线程B 抢占到CPU后开始执行，执行到 [if (p == NULL)]时， 发现 p 是 NULL，执行 if 内的代码
4. 线程B 执行完后，线程A 重新被调度，继续执行 if 的代码   
     其实此时由于线程B 已经执行完，p 已经不是 NULL了，线程A 可能会破坏线程B 已经完成的处理，导致数据不一致

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。

##### 4.2.1 按CPU分配的优势

按CPU来分配数据主要有2个优点：

1. 最直接的效果就是减少了对数据的锁，提高了系统的性能
2. 由于每个CPU有自己的数据，所以处理器切换时可以大大减少缓存失效的几率 (**\*注1**)

**注1**：如果一个处理器操作某个数据，而这个数据在另一个处理器的缓存中时，那么存放这个数据的那个

处理器必须清理或刷新自己的缓存。持续的缓存失效成为缓存抖动，对系统性能影响很大。

##### 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。

其实也就是 DECLARE\_PER\_CPU 中多了个 extern 的关键字

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 \*/

通过 get\_cpu\_var 和 put\_cpu\_var 的代码，我们可以发现其中有禁止和激活内核抢占的函数。

相关代码在 <linux/percpu.h> 中

#define get\_cpu\_var(var) (\*({ \

extern int simple\_identifier\_##var(void); \

preempt\_disable();/\* 这句就是禁止当前处理器上的内核抢占 \*/ \

&\_\_get\_cpu\_var(var); }))

#define put\_cpu\_var(var) preempt\_enable() /\* 这句就是激活当前处理器上的内核抢占 \*/

##### 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>

[复制代码](javascript:void(0);)

/\* 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))

[复制代码](javascript:void(0);)

动态分配的一个使用例子如下：

[复制代码](javascript:void(0);)

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);

[复制代码](javascript:void(0);)

### 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 |

# [（十三）- 虚拟文件系统](http://www.cnblogs.com/wang_yb/p/3144291.html)

虚拟文件系统（VFS）是linux内核和具体I/O设备之间的封装的一层共通访问接口，通过这层接口，linux内核可以以同一的方式访问各种I/O设备。

虚拟文件系统本身是linux内核的一部分，是纯软件的东西，并不需要任何硬件的支持。

**主要内容：**

* 虚拟文件系统的作用
* 虚拟文件系统的4个主要对象
* 文件系统相关的数据结构
* 进程相关的数据结构
* 小结

### 1. 虚拟文件系统的作用

虚拟文件系统(VFS)是linux内核和存储设备之间的抽象层，主要有以下好处。

- 简化了应用程序的开发：应用通过统一的系统调用访问各种存储介质

- 简化了新文件系统加入内核的过程：新文件系统只要实现VFS的各个接口即可，不需要修改内核部分

### 2. 虚拟文件系统的4个主要对象

虚拟文件中的4个主要对象，具体每个对象的含义参见如下的详细介绍。

#### 2.1 超级块

超级块(super\_block)主要存储文件系统相关的信息，这是个针对文件系统级别的概念。

它一般存储在磁盘的特定扇区中，但是对于那些基于内存的文件系统(比如proc,sysfs)，超级块是在使用时创建在内存中的。

超级块的定义在：<linux/fs.h>

[复制代码](javascript:void(0);)

/\*

\* 超级块结构中定义的字段非常多，

\* 这里只介绍一些重要的属性

\*/

struct super\_block {

struct list\_head s\_list; /\* 指向所有超级块的链表 \*/

const struct super\_operations \*s\_op; /\* 超级块方法 \*/

struct dentry \*s\_root; /\* 目录挂载点 \*/

struct mutex s\_lock; /\* 超级块信号量 \*/

int s\_count; /\* 超级块引用计数 \*/

struct list\_head s\_inodes; /\* inode链表 \*/

struct mtd\_info \*s\_mtd; /\* 存储磁盘信息 \*/

fmode\_t s\_mode; /\* 安装权限 \*/

};

/\*

\* 其中的 s\_op 中定义了超级块的操作方法

\* 这里只介绍一些相对重要的函数

\*/

struct super\_operations {

struct inode \*(\*alloc\_inode)(struct super\_block \*sb); /\* 创建和初始化一个索引节点对象 \*/

void (\*destroy\_inode)(struct inode \*); /\* 释放给定的索引节点 \*/

void (\*dirty\_inode) (struct inode \*); /\* VFS在索引节点被修改时会调用这个函数 \*/

int (\*write\_inode) (struct inode \*, int); /\* 将索引节点写入磁盘，wait表示写操作是否需要同步 \*/

void (\*drop\_inode) (struct inode \*); /\* 最后一个指向索引节点的引用被删除后，VFS会调用这个函数 \*/

void (\*delete\_inode) (struct inode \*); /\* 从磁盘上删除指定的索引节点 \*/

void (\*put\_super) (struct super\_block \*); /\* 卸载文件系统时由VFS调用，用来释放超级块 \*/

void (\*write\_super) (struct super\_block \*); /\* 用给定的超级块更新磁盘上的超级块 \*/

int (\*sync\_fs)(struct super\_block \*sb, int wait); /\* 使文件系统中的数据与磁盘上的数据同步 \*/

int (\*statfs) (struct dentry \*, struct kstatfs \*); /\* VFS调用该函数获取文件系统状态 \*/

int (\*remount\_fs) (struct super\_block \*, int \*, char \*); /\* 指定新的安装选项重新安装文件系统时，VFS会调用该函数 \*/

void (\*clear\_inode) (struct inode \*); /\* VFS调用该函数释放索引节点，并清空包含相关数据的所有页面 \*/

void (\*umount\_begin) (struct super\_block \*); /\* VFS调用该函数中断安装操作 \*/

};

[复制代码](javascript:void(0);)

#### 2.2 索引节点

索引节点是VFS中的核心概念，它包含内核在操作文件或目录时需要的全部信息。

一个索引节点代表文件系统中的一个文件(这里的文件不仅是指我们平时所认为的普通的文件，还包括目录，特殊设备文件等等)。

索引节点和超级块一样是实际存储在磁盘上的，当被应用程序访问到时才会在内存中创建。

索引节点定义在：<linux/fs.h>

[复制代码](javascript:void(0);)

/\*

\* 索引节点结构中定义的字段非常多，

\* 这里只介绍一些重要的属性

\*/

struct inode {

struct hlist\_node i\_hash; /\* 散列表，用于快速查找inode \*/

struct list\_head i\_list; /\* 索引节点链表 \*/

struct list\_head i\_sb\_list; /\* 超级块链表超级块 \*/

struct list\_head i\_dentry; /\* 目录项链表 \*/

unsigned long i\_ino; /\* 节点号 \*/

atomic\_t i\_count; /\* 引用计数 \*/

unsigned int i\_nlink; /\* 硬链接数 \*/

uid\_t i\_uid; /\* 使用者id \*/

gid\_t i\_gid; /\* 使用组id \*/

struct timespec i\_atime; /\* 最后访问时间 \*/

struct timespec i\_mtime; /\* 最后修改时间 \*/

struct timespec i\_ctime; /\* 最后改变时间 \*/

const struct inode\_operations \*i\_op; /\* 索引节点操作函数 \*/

const struct file\_operations \*i\_fop; /\* 缺省的索引节点操作 \*/

struct super\_block \*i\_sb; /\* 相关的超级块 \*/

struct address\_space \*i\_mapping; /\* 相关的地址映射 \*/

struct address\_space i\_data; /\* 设备地址映射 \*/

unsigned int i\_flags; /\* 文件系统标志 \*/

void \*i\_private; /\* fs 私有指针 \*/

};

/\*

\* 其中的 i\_op 中定义了索引节点的操作方法

\* 这里只介绍一些相对重要的函数

\*/

struct inode\_operations {

/\* 为dentry对象创造一个新的索引节点 \*/

int (\*create) (struct inode \*,struct dentry \*,int, struct nameidata \*);

/\* 在特定文件夹中寻找索引节点，该索引节点要对应于dentry中给出的文件名 \*/

struct dentry \* (\*lookup) (struct inode \*,struct dentry \*, struct nameidata \*);

/\* 创建硬链接 \*/

int (\*link) (struct dentry \*,struct inode \*,struct dentry \*);

/\* 从一个符号链接查找它指向的索引节点 \*/

void \* (\*follow\_link) (struct dentry \*, struct nameidata \*);

/\* 在 follow\_link调用之后，该函数由VFS调用进行清除工作 \*/

void (\*put\_link) (struct dentry \*, struct nameidata \*, void \*);

/\* 该函数由VFS调用，用于修改文件的大小 \*/

void (\*truncate) (struct inode \*);

};

[复制代码](javascript:void(0);)

#### 2.3 目录项

和超级块和索引节点不同，目录项并不是实际存在于磁盘上的。

在使用的时候在内存中创建目录项对象，其实通过索引节点已经可以定位到指定的文件，

但是索引节点对象的属性非常多，在查找，比较文件时，直接用索引节点效率不高，所以引入了目录项的概念。

路径中的每个部分都是一个目录项，比如路径： /mnt/cdrom/foo/bar 其中包含5个目录项，/ mnt cdrom foo bar

每个目录项对象都有3种状态：被使用，未使用和负状态

- 被使用：对应一个有效的索引节点，并且该对象由一个或多个使用者

- 未使用：对应一个有效的索引节点，但是VFS当前并没有使用这个目录项

- 负状态：没有对应的有效索引节点（可能索引节点被删除或者路径不存在了）

目录项的目的就是提高文件查找，比较的效率，所以访问过的目录项都会缓存在slab中。

slab中缓存的名称一般就是 dentry，可以通过如下命令查看：

[wangyubin@localhost kernel]$ sudo cat /proc/slabinfo | grep dentry

dentry 212545 212625 192 21 1 : tunables 0 0 0 : slabdata 10125 10125 0

目录项定义在：<linux/dcache.h>

[复制代码](javascript:void(0);)

/\* 目录项对象结构 \*/

struct dentry {

atomic\_t d\_count; /\* 使用计数 \*/

unsigned int d\_flags; /\* 目录项标识 \*/

spinlock\_t d\_lock; /\* 单目录项锁 \*/

int d\_mounted; /\* 是否登录点的目录项 \*/

struct inode \*d\_inode; /\* 相关联的索引节点 \*/

struct hlist\_node d\_hash; /\* 散列表 \*/

struct dentry \*d\_parent; /\* 父目录的目录项对象 \*/

struct qstr d\_name; /\* 目录项名称 \*/

struct list\_head d\_lru; /\* 未使用的链表 \*/

/\*

\* d\_child and d\_rcu can share memory

\*/

union {

struct list\_head d\_child; /\* child of parent list \*/

struct rcu\_head d\_rcu;

} d\_u;

struct list\_head d\_subdirs; /\* 子目录链表 \*/

struct list\_head d\_alias; /\* 索引节点别名链表 \*/

unsigned long d\_time; /\* 重置时间 \*/

const struct dentry\_operations \*d\_op; /\* 目录项操作相关函数 \*/

struct super\_block \*d\_sb; /\* 文件的超级块 \*/

void \*d\_fsdata; /\* 文件系统特有数据 \*/

unsigned char d\_iname[DNAME\_INLINE\_LEN\_MIN]; /\* 短文件名 \*/

};

/\* 目录项相关操作函数 \*/

struct dentry\_operations {

/\* 该函数判断目录项对象是否有效。VFS准备从dcache中使用一个目录项时会调用这个函数 \*/

int (\*d\_revalidate)(struct dentry \*, struct nameidata \*);

/\* 为目录项对象生成hash值 \*/

int (\*d\_hash) (struct dentry \*, struct qstr \*);

/\* 比较 qstr 类型的2个文件名 \*/

int (\*d\_compare) (struct dentry \*, struct qstr \*, struct qstr \*);

/\* 当目录项对象的 d\_count 为0时，VFS调用这个函数 \*/

int (\*d\_delete)(struct dentry \*);

/\* 当目录项对象将要被释放时，VFS调用该函数 \*/

void (\*d\_release)(struct dentry \*);

/\* 当目录项对象丢失其索引节点时（也就是磁盘索引节点被删除了），VFS会调用该函数 \*/

void (\*d\_iput)(struct dentry \*, struct inode \*);

char \*(\*d\_dname)(struct dentry \*, char \*, int);

};

[复制代码](javascript:void(0);)

#### 2.4 文件对象

文件对象表示进程已打开的文件，从用户角度来看，我们在代码中操作的就是一个文件对象。

文件对象反过来指向一个目录项对象（目录项反过来指向一个索引节点）

其实只有目录项对象才表示一个已打开的实际文件，虽然一个文件对应的文件对象不是唯一的，但其对应的索引节点和目录项对象却是唯一的。

文件对象的定义在: <linux/fs.h>

[复制代码](javascript:void(0);)

/\*

\* 文件对象结构中定义的字段非常多，

\* 这里只介绍一些重要的属性

\*/

struct file {

union {

struct list\_head fu\_list; /\* 文件对象链表 \*/

struct rcu\_head fu\_rcuhead; /\* 释放之后的RCU链表 \*/

} f\_u;

struct path f\_path; /\* 包含的目录项 \*/

const struct file\_operations \*f\_op; /\* 文件操作函数 \*/

atomic\_long\_t f\_count; /\* 文件对象引用计数 \*/

};

/\*

\* 其中的 f\_op 中定义了文件对象的操作方法

\* 这里只介绍一些相对重要的函数

\*/

struct file\_operations {

/\* 用于更新偏移量指针,由系统调用lleek()调用它 \*/

loff\_t (\*llseek) (struct file \*, loff\_t, int);

/\* 由系统调用read()调用它 \*/

ssize\_t (\*read) (struct file \*, char \_\_user \*, size\_t, loff\_t \*);

/\* 由系统调用write()调用它 \*/

ssize\_t (\*write) (struct file \*, const char \_\_user \*, size\_t, loff\_t \*);

/\* 由系统调用 aio\_read() 调用它 \*/

ssize\_t (\*aio\_read) (struct kiocb \*, const struct iovec \*, unsigned long, loff\_t);

/\* 由系统调用 aio\_write() 调用它 \*/

ssize\_t (\*aio\_write) (struct kiocb \*, const struct iovec \*, unsigned long, loff\_t);

/\* 将给定文件映射到指定的地址空间上,由系统调用 mmap 调用它 \*/

int (\*mmap) (struct file \*, struct vm\_area\_struct \*);

/\* 创建一个新的文件对象,并将它和相应的索引节点对象关联起来 \*/

int (\*open) (struct inode \*, struct file \*);

/\* 当已打开文件的引用计数减少时,VFS调用该函数 \*/

int (\*flush) (struct file \*, fl\_owner\_t id);

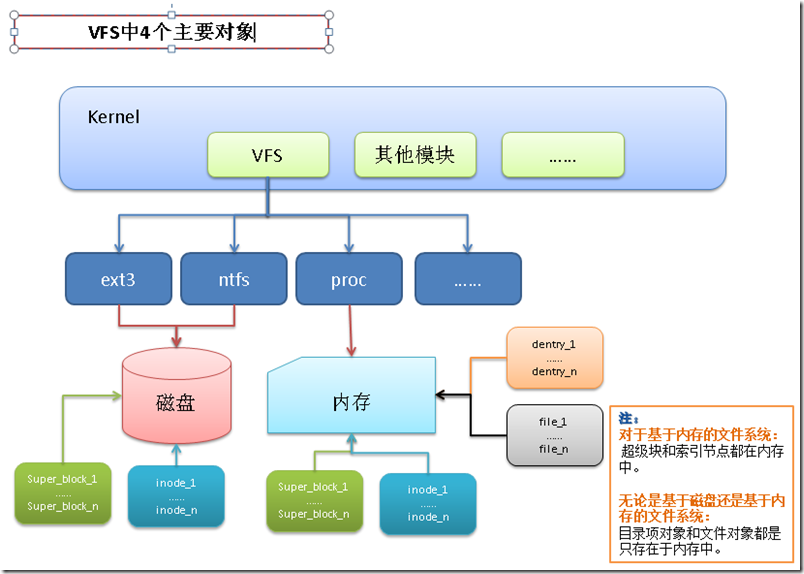
};

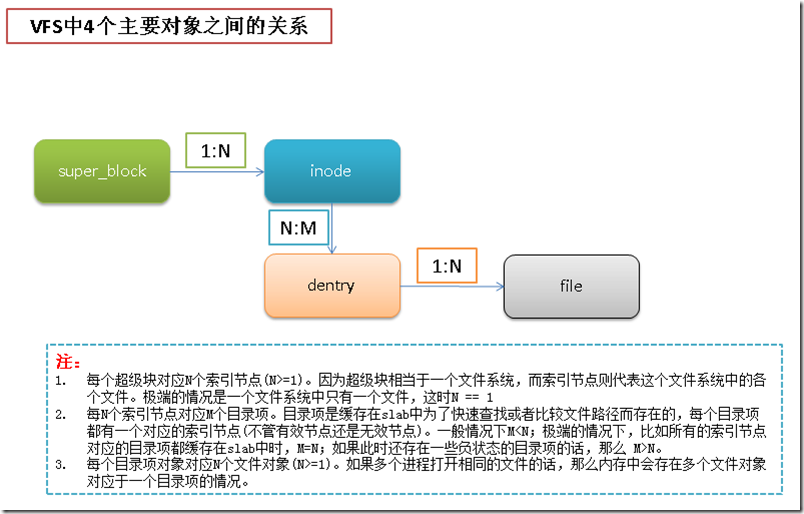
[复制代码](javascript:void(0);)

#### 2.5 四个对象之间关系图

上面分别介绍了4种对象分别的属性和方法,下面用图来展示这4个对象的和VFS之间关系以及4个对象之间的关系。

(这个图是根据我自己的理解画出来的,如果由错误请帮忙指出,谢谢!)

[](https://images0.cnblogs.com/blog/83005/201306/19154220-7d6965bfe1f942afa6016e21dcfb23d3.png)

[](https://images0.cnblogs.com/blog/83005/201306/19154226-f2eb0b2cf5c540a09faa88a5a5af68df.png)

### 3. 文件系统相关的数据结构

处理上面4个主要的对象之外，VFS中还有2个专门针对文件系统的2个对象，

- struct file\_system\_type: 用来描述文件系统的类型（比如ext3,ntfs等等）

- struct vfsmount        : 描述一个安装文件系统的实例

file\_system\_type 结构体位于：<linux/fs.h>

[复制代码](javascript:void(0);)

struct file\_system\_type {

const char \*name; /\* 文件系统名称 \*/

int fs\_flags; /\* 文件系统类型标志 \*/

/\* 从磁盘中读取超级块,并且在文件系统被安装时,在内存中组装超级块对象 \*/

int (\*get\_sb) (struct file\_system\_type \*, int,

const char \*, void \*, struct vfsmount \*);

/\* 终止访问超级块 \*/

void (\*kill\_sb) (struct super\_block \*);

struct module \*owner; /\* 文件系统模块 \*/

struct file\_system\_type \* next; /\* 链表中下一个文件系统类型 \*/

struct list\_head fs\_supers; /\* 超级块对象链表 \*/

/\* 下面都是运行时的锁 \*/

struct lock\_class\_key s\_lock\_key;

struct lock\_class\_key s\_umount\_key;

struct lock\_class\_key i\_lock\_key;

struct lock\_class\_key i\_mutex\_key;

struct lock\_class\_key i\_mutex\_dir\_key;

struct lock\_class\_key i\_alloc\_sem\_key;

};

[复制代码](javascript:void(0);)

每种文件系统,不管由多少个实例安装到系统中,还是根本没有安装到系统中,都只有一个 file\_system\_type 结构。

当文件系统被实际安装时，会在安装点创建一个 vfsmount 结构体。

结构体代表文件系统的实例，也就是文件系统被安装几次，就会创建几个 vfsmount

vfsmount 的定义参见：<linux/mount.h>

[复制代码](javascript:void(0);)

struct vfsmount {

struct list\_head mnt\_hash; /\* 散列表 \*/

struct vfsmount \*mnt\_parent; /\* 父文件系统，也就是要挂载到哪个文件系统 \*/

struct dentry \*mnt\_mountpoint; /\* 安装点的目录项 \*/

struct dentry \*mnt\_root; /\* 该文件系统的根目录项 \*/

struct super\_block \*mnt\_sb; /\* 该文件系统的超级块 \*/

struct list\_head mnt\_mounts; /\* 子文件系统链表 \*/

struct list\_head mnt\_child; /\* 子文件系统链表 \*/

int mnt\_flags; /\* 安装标志 \*/

/\* 4 bytes hole on 64bits arches \*/

const char \*mnt\_devname; /\* 设备文件名 e.g. /dev/dsk/hda1 \*/

struct list\_head mnt\_list; /\* 描述符链表 \*/

struct list\_head mnt\_expire; /\* 到期链表的入口 \*/

struct list\_head mnt\_share; /\* 共享安装链表的入口 \*/

struct list\_head mnt\_slave\_list;/\* 从安装链表 \*/

struct list\_head mnt\_slave; /\* 从安装链表的入口 \*/

struct vfsmount \*mnt\_master; /\* 从安装链表的主人 \*/

struct mnt\_namespace \*mnt\_ns; /\* 相关的命名空间 \*/

int mnt\_id; /\* 安装标识符 \*/

int mnt\_group\_id; /\* 组标识符 \*/

/\*

\* We put mnt\_count & mnt\_expiry\_mark at the end of struct vfsmount

\* to let these frequently modified fields in a separate cache line

\* (so that reads of mnt\_flags wont ping-pong on SMP machines)

\*/

atomic\_t mnt\_count; /\* 使用计数 \*/

int mnt\_expiry\_mark; /\* 如果标记为到期，则为 True \*/

int mnt\_pinned; /\* "钉住"进程计数 \*/

int mnt\_ghosts; /\* "镜像"引用计数 \*/

#ifdef CONFIG\_SMP

int \*mnt\_writers; /\* 写者引用计数 \*/

#else

int mnt\_writers; /\* 写者引用计数 \*/

#endif

};

[复制代码](javascript:void(0);)

### 4. 进程相关的数据结构

以上介绍的都是在内核角度看到的 VFS 各个结构，所以结构体中包含的属性非常多。

而从进程的角度来看的话，大多数时候并不需要那么多的属性，所有VFS通过以下3个结构体和进程紧密联系在一起。

- struct files\_struct  ：由进程描述符中的 files 目录项指向，所有与单个进程相关的信息(比如打开的文件和文件描述符)都包含在其中。

- struct fs\_struct     ：由进程描述符中的 fs 域指向，包含文件系统和进程相关的信息。

- struct mmt\_namespace ：由进程描述符中的 mmt\_namespace 域指向。

struct files\_struct 位于：<linux/fdtable.h>

[复制代码](javascript:void(0);)

struct files\_struct {

atomic\_t count; /\* 使用计数 \*/

struct fdtable \*fdt; /\* 指向其他fd表的指针 \*/

struct fdtable fdtab;/\* 基 fd 表 \*/

spinlock\_t file\_lock \_\_\_\_cacheline\_aligned\_in\_smp; /\* 单个文件的锁 \*/

int next\_fd; /\* 缓存下一个可用的fd \*/

struct embedded\_fd\_set close\_on\_exec\_init; /\* exec()时关闭的文件描述符链表 \*/

struct embedded\_fd\_set open\_fds\_init; /\* 打开的文件描述符链表 \*/

struct file \* fd\_array[NR\_OPEN\_DEFAULT]; /\* 缺省的文件对象数组 \*/

};

[复制代码](javascript:void(0);)

struct fs\_struct 位于：<linux/fs\_struct.h>

[复制代码](javascript:void(0);)

struct fs\_struct {

int users; /\* 用户数目 \*/

rwlock\_t lock; /\* 保护结构体的读写锁 \*/

int umask; /\* 掩码 \*/

int in\_exec; /\* 当前正在执行的文件 \*/

struct path root, pwd; /\* 根目录路径和当前工作目录路径 \*/

};

[复制代码](javascript:void(0);)

struct mmt\_namespace 位于：<linux/mmt\_namespace.h>

但是在2.6内核之后似乎没有这个结构体了，而是用 struct nsproxy 来代替。

以下是 struct task\_struct 结构体中关于文件系统的3个属性。

struct task\_struct 的定义位于：<linux/sched.h>

/\* filesystem information \*/

struct fs\_struct \*fs;

/\* open file information \*/

struct files\_struct \*files;

/\* namespaces \*/

struct nsproxy \*nsproxy;

### 5. 小结

VFS 统一了文件系统的实现框架，使得在linux上实现新文件系统的工作变得简单。

目前linux内核中已经支持60多种文件系统，具体支持的文件系统可以查看 内核源码 fs 文件夹下的内容。

# [（十四）- 块I/O层](http://www.cnblogs.com/wang_yb/p/3299092.html)

最近太忙，居然过了2个月才更新第十四章。。。。

**主要内容：**

* 块设备简介
* 内核访问块设备的方法
* 内核I/O调度程序

### ****1. 块设备简介****

I/O设备主要有2类:

* 字符设备：只能顺序读写设备中的内容，比如 串口设备，键盘
* 块设备：能够随机读写设备中的内容，比如 硬盘，U盘

字符设备由于只能顺序访问，所以应用场景也不多，这篇文章主要讨论块设备。

块设备是随机访问的，所以块设备在不同的应用场景中存在很大的优化空间。

块设备中最重要的一个概念就是块设备的最小寻址单元。

块设备的最小寻址单元就是扇区，扇区的大小是2的整数倍，一般是 512字节。

扇区是物理上的最小寻址单元，而逻辑上的最小寻址单元是块。

为了便于文件系统管理，块的大小一般是扇区的整数倍，并且小于等于页的大小。

查看扇区和I/O块的方法：

[复制代码](javascript:void(0);)

[wangyubin@localhost]$ sudo fdisk -l

WARNING: GPT (GUID Partition Table) detected on '/dev/sda'! The util fdisk doesn't support GPT. Use GNU Parted.

Disk /dev/sda: 500.1 GB, 500107862016 bytes, 976773168 sectors

Units = sectors of 1 \* 512 = 512 bytes

Sector size (logical/physical): 512 bytes / 4096 bytes

I/O size (minimum/optimal): 4096 bytes / 4096 bytes

Disk identifier: 0x00000000

[复制代码](javascript:void(0);)

上面的 Sector size 就是扇区的值，I/O size就是 块的值

从上面显示的结果，我们发现有个奇怪的地方，扇区的大小有2个值，逻辑大小是 512字节，而物理大小却是 4096字节。

其实逻辑大小 512字节是为了兼容以前的软件应用，而实际物理大小 4096字节是由于硬盘空间越来越大导致的。

具体的来龙去脉请参考：[4KB扇区的原因](http://www.ibm.com/developerworks/cn/linux/l-4kb-sector-disks/)

### ****2. 内核访问块设备的方法****

内核通过文件系统访问块设备时，需要先把块读入到内存中。所以文件系统为了管理块设备，必须管理[块]和内存页之间的映射。

内核中有2种方法来管理 [**块**] 和内存页之间的映射。

* 缓冲区和缓冲区头
* bio

#### 2.1 缓冲区和缓冲区头

每个 [**块**] 都是一个缓冲区，同时对每个 [**块**] 都定义一个缓冲区头来描述它。

由于 [**块**] 的大小是小于内存页的大小的，所以每个内存页会包含一个或者多个 [**块**]

缓冲区头定义在 <linux/buffer\_head.h>: include/linux/buffer\_head.h

[复制代码](javascript:void(0);)

struct buffer\_head {

unsigned long b\_state; /\* 表示缓冲区状态 \*/

struct buffer\_head \*b\_this\_page;/\* 当前页中缓冲区 \*/

struct page \*b\_page; /\* 当前缓冲区所在内存页 \*/

sector\_t b\_blocknr; /\* 起始块号 \*/

size\_t b\_size; /\* buffer在内存中的大小 \*/

char \*b\_data; /\* 块映射在内存页中的数据 \*/

struct block\_device \*b\_bdev; /\* 关联的块设备 \*/

bh\_end\_io\_t \*b\_end\_io; /\* I/O完成方法 \*/

void \*b\_private; /\* 保留的 I/O 完成方法 \*/

struct list\_head b\_assoc\_buffers; /\* 关联的其他缓冲区 \*/

struct address\_space \*b\_assoc\_map; /\* 相关的地址空间 \*/

atomic\_t b\_count; /\* 引用计数 \*/

};

[复制代码](javascript:void(0);)

整个 buffer\_head 结构体中的字段是减少过的，以前的内核中字段更多。

各个字段的含义通过注释都很明了，只有 b\_state 字段比较复杂，它涵盖了缓冲区可能的各种状态。

[复制代码](javascript:void(0);)

enum bh\_state\_bits {

BH\_Uptodate, /\* 包含可用数据 \*/

BH\_Dirty, /\* 该缓冲区是脏的(说明缓冲的内容比磁盘中的内容新，需要回写磁盘) \*/

BH\_Lock, /\* 该缓冲区正在被I/O使用，锁住以防止并发访问 \*/

BH\_Req, /\* 该缓冲区有I/O请求操作 \*/

BH\_Uptodate\_Lock,/\* 由内存页中的第一个缓冲区使用，使得该页中的其他缓冲区 \*/

BH\_Mapped, /\* 该缓冲区是映射到磁盘块的可用缓冲区 \*/

BH\_New, /\* 缓冲区是通过 get\_block() 刚刚映射的，尚且不能访问 \*/

BH\_Async\_Read, /\* 该缓冲区正通过 end\_buffer\_async\_read() 被异步I/O读操作使用 \*/

BH\_Async\_Write, /\* 该缓冲区正通过 end\_buffer\_async\_read() 被异步I/O写操作使用 \*/

BH\_Delay, /\* 缓冲区还未和磁盘关联 \*/

BH\_Boundary, /\* 该缓冲区处于连续块区的边界，下一个块不在连续 \*/

BH\_Write\_EIO, /\* 该缓冲区在写的时候遇到 I/O 错误 \*/

BH\_Ordered, /\* 顺序写 \*/

BH\_Eopnotsupp, /\* 该缓冲区发生 “不被支持” 错误 \*/

BH\_Unwritten, /\* 该缓冲区在磁盘上的位置已经被申请，但还有实际写入数据 \*/

BH\_Quiet, /\* 该缓冲区禁止错误 \*/

BH\_PrivateStart,/\* 不是表示状态，分配给其他实体的私有数据区的第一个bit \*/

};

[复制代码](javascript:void(0);)

在2.6之前的内核中，主要就是通过缓冲区头来管理 [块] 和内存之间的映射的。

用缓冲区头来管理内核的 I/O 操作主要存在以下2个弊端，所以在2.6开始的内核中，缓冲区头的作用大大降低了。

- 弊端 1

对内核而言，操作内存页是最为简便和高效的，所以如果通过缓冲区头来操作的话（缓冲区 即[块]在内存中映射，可能比页面要小），效率低下。

而且每个 [块] 对应一个缓冲区头的话，导致内存的利用率降低（缓冲区头包含的字段非常多）

- 弊端 2

每个缓冲区头只能表示一个 [块]，所以内核在处理大数据时，会分解为对一个个小的 [块] 的操作，造成不必要的负担和空间浪费。

#### 2.2 bio

bio结构体的出现就是为了改善上面缓冲区头的2个弊端，它表示了一次 I/O 操作所涉及到的所有内存页。

[复制代码](javascript:void(0);)

/\*

\* I/O 操作的主要单元，针对 I/O块和更低级的层 (ie drivers and

\* stacking drivers)

\*/

struct bio {

sector\_t bi\_sector; /\* 磁盘上相关扇区 \*/

struct bio \*bi\_next; /\* 请求列表 \*/

struct block\_device \*bi\_bdev; /\* 相关的块设备 \*/

unsigned long bi\_flags; /\* 状态和命令标志 \*/

unsigned long bi\_rw; /\* 读还是写 \*/

unsigned short bi\_vcnt; /\* bio\_vecs的数目 \*/

unsigned short bi\_idx; /\* bio\_io\_vect的当前索引 \*/

/\* Number of segments in this BIO after

\* physical address coalescing is performed.

\* 结合后的片段数目

\*/

unsigned int bi\_phys\_segments;

unsigned int bi\_size; /\* 剩余 I/O 计数 \*/

/\*

\* To keep track of the max segment size, we account for the

\* sizes of the first and last mergeable segments in this bio.

\* 第一个和最后一个可合并的段的大小

\*/

unsigned int bi\_seg\_front\_size;

unsigned int bi\_seg\_back\_size;

unsigned int bi\_max\_vecs; /\* bio\_vecs数目上限 \*/

unsigned int bi\_comp\_cpu; /\* 结束CPU \*/

atomic\_t bi\_cnt; /\* 使用计数 \*/

struct bio\_vec \*bi\_io\_vec; /\* bio\_vec 链表 \*/

bio\_end\_io\_t \*bi\_end\_io; /\* I/O 完成方法 \*/

void \*bi\_private; /\* bio结构体创建者的私有方法 \*/

#if defined(CONFIG\_BLK\_DEV\_INTEGRITY)

struct bio\_integrity\_payload \*bi\_integrity; /\* data integrity \*/

#endif

bio\_destructor\_t \*bi\_destructor; /\* bio撤销方法 \*/

/\*

\* We can inline a number of vecs at the end of the bio, to avoid

\* double allocations for a small number of bio\_vecs. This member

\* MUST obviously be kept at the very end of the bio.

\* 内嵌在结构体末尾的 bio 向量，主要为了防止出现二次申请少量的 bio\_vecs

\*/

struct bio\_vec bi\_inline\_vecs[0];

};

[复制代码](javascript:void(0);)

几个重要字段说明：

* bio 结构体表示正在执行的 I/O 操作相关的信息。
* bio\_io\_vec 链表表示当前 I/O 操作涉及到的内存页
* bio\_vec 结构体表示 I/O 操作使用的片段
* bi\_vcnt bi\_io\_vec链表中bi\_vec的个数
* bi\_idx 当前的 bi\_vec片段，通过 bi\_vcnt（总数）和 bi\_idx（当前数），就可以跟踪当前 I/O 操作的进度

bio\_vec 结构体很简单，定义如下：

struct bio\_vec {

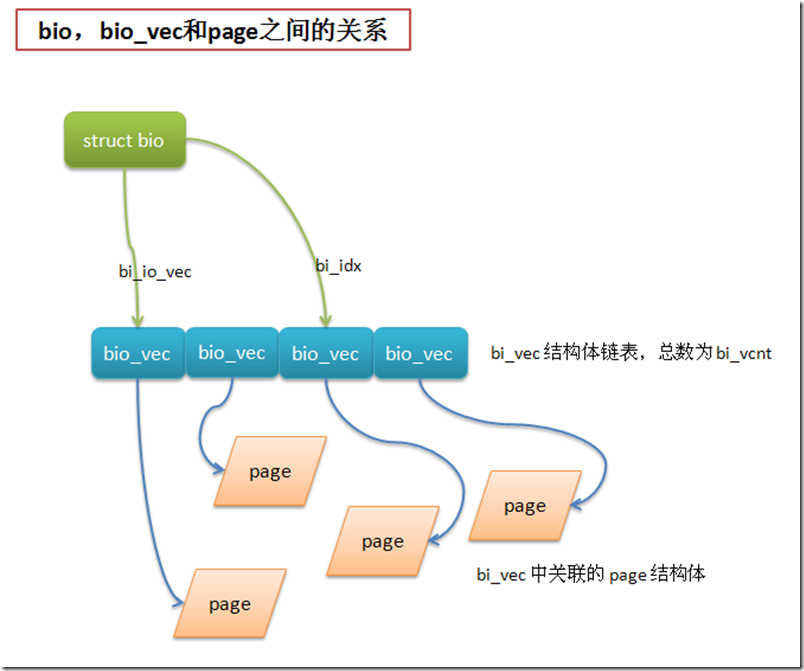
struct page \*bv\_page; /\* 对应的物理页 \*/

unsigned int bv\_len; /\* 缓冲区大小 \*/

unsigned int bv\_offset; /\* 缓冲区开始的位置 \*/

};

每个 bio\_vec 都是对应一个页面，从而保证内核能够方便高效的完成 I/O 操作

[](https://images0.cnblogs.com/blog/83005/201309/03170237-f24ead41f0284c6a935c50ba966d0f47.png)

#### 2.3 2种方法的对比

缓冲区头和bio并不是相互矛盾的，bio只是缓冲区头的一种改善，将以前缓冲区头完成的一部分工作移到bio中来完成。

bio中对应的是内存中的一个个页，而缓冲区头对应的是磁盘中的一个块。

对内核来说，配合使用bio和缓冲区头 比 只使用缓冲区头更加的方便高效。

bio相当于在缓冲区上又封装了一层，使得内核在 I/O操作时只要针对一个或多个内存页即可，不用再去管理磁盘块的部分。

使用bio结构体还有以下好处：

* bio结构体很容易处理高端内存，因为它处理的是内存页而不是直接指针
* bio结构体既可以代表普通页I/O，也可以代表直接I/O
* bio结构体便于执行分散-集中（矢量化的）块I/O操作，操作中的数据可以取自多个物理页面

### ****3. 内核I/O调度程序****

缓冲区头和bio都是内核处理一个具体I/O操作时涉及的概念。

但是内核除了要完成I/O操作以外，还要调度好所有I/O操作请求，尽量确保每个请求能有个合理的响应时间。

下面就是目前内核中已有的一些 I/O 调度算法。

#### 3.1 linus电梯

为了保证磁盘寻址的效率，一般会尽量让磁头向一个方向移动，等到头了再反过来移动，这样可以缩短所有请求的磁盘寻址总时间。

磁头的移动有点类似于电梯，所有这个 I/O 调度算法也叫电梯调度。

linux中的第一个电梯调度算法就是 linus本人所写的，所有也叫做 linus 电梯。

linus电梯调度主要是对I/O请求进行合并和排序。

当一个新请求加入I/O请求队列时，可能会发生以下4种操作:

1. 如果队列中已存在一个对相邻磁盘扇区操作的请求，那么新请求将和这个已存在的请求合并成一个请求
2. 如果队列中存在一个驻留时间过长的请求，那么新请求之间查到队列尾部，防止旧的请求发生饥饿
3. 如果队列中已扇区方向为序存在合适的插入位置，那么新请求将被插入该位置，保证队列中的请求是以被访问磁盘物理位置为序进行排列的
4. 如果队列中不存在合适的请求插入位置，请求将被插入到队列尾部

linus电梯调度程序在2.6版的内核中被其他调度程序所取代了。

#### 3.2 最终期限I/O调度

linus电梯调度主要考虑了系统的全局吞吐量，对于个别的I/O请求，还是有可能造成饥饿现象。

而且读写请求的响应时间要求也是不一样的，一般来说，写请求的响应时间要求不高，写请求可以和提交它的应用程序异步执行，

但是读请求一般和提交它的应用程序时同步执行，应用程序等获取到读的数据后才会接着往下执行。

因此在 linus 电梯调度程序中，还可能造成 写-饥饿-读（wirtes-starving-reads）这种特殊问题。

为了尽量公平的对待所有请求，同时尽量保证读请求的响应时间，提出了最终期限I/O调度算法。

最终期限I/O调度 算法给每个请求设置了超时时间，默认情况下，读请求的超时时间500ms，写请求的超时时间是5s

但一个新请求加入到I/O请求队列时，最终期限I/O调度和linus电梯调度相比，多出了以下操作:

1. 新请求加入到 排序队列(order-FIFO)，加入的方法类似 linus电梯新请求加入的方法
2. 根据新请求的类型，将其加入 读队列(read-FIFO) 或者写队列(wirte-FIFO) 的尾部（读写队列是按加入时间排序的，所以新请求都是加到尾部）
3. 调度程序首先判断 读，写队列头的请求是否超时，如果超时，从读，写队列头取出请求，加入到派发队列(dispatch-FIFO)
4. 如果没有超时请求，从 排序队列(order-FIFO)头取出一个请求加入到 派发队列(dispatch-FIFO)
5. 派发队列(dispatch-FIFO)按顺序将请求提交到磁盘驱动，完成I/O操作

最终期限I/O调度 算法也不能严格保证响应时间，但是它可以保证不会发生请求在明显超时的情况下仍得不到执行。

最终期限I/O调度 的实现参见: block/deadline-iosched.c

#### 3.3 预测I/O调度

最终期限I/O调度算法优先考虑读请求的响应时间，但系统处于写操作繁重的状态时，会大大降低系统的吞吐量。

因为读请求的超时时间比较短，所以每次有读请求时，都会打断写请求，让磁盘寻址到读的位置，完成读操作后再回来继续写。

这种做法保证读请求的响应速度，却损害了系统的全局吞吐量（磁头先去读再回来写，发生了2次寻址操作）

预测I/O调度算法是为了解决上述问题而提出的，它是基于最终期限I/O调度算法的。

但有一个新请求加入到I/O请求队列时，预测I/O调度与最终期限I/O调度相比，多了以下操作:

1. 新的读请求提交后，并不立即进行请求处理，而是有意等待片刻(默认是6ms)
2. 等待期间如果有其他对磁盘相邻位置进行读操作的读请求加入，会立刻处理这些读请求
3. 等待期间如果没有其他读请求加入，那么等待时间相当于浪费掉
4. 等待时间结束后，继续执行以前剩下的请求

预测I/O调度算法中最重要的是保证等待期间不要浪费，也就是提高预测的准确性，

目前这种预测是依靠一系列的启发和统计工作，预测I/O调度程序会跟踪并统计每个应用程序的I/O操作习惯，以便正确预测应用程序的读写行为。

如果预测的准确率足够高，那么预测I/O调度和最终期限I/O调度相比，既能提高读请求的响应时间，又能提高系统吞吐量。

预测I/O调度的实现参见: block/as-iosched.c

**注**:预测I/O调度是linux内核中缺省的调度程序。

#### 3.4 完全公正的排队I/O调度

完全公正的排队(Complete Fair Queuing, CFQ)I/O调度 是为专有工作负荷设计的，它和之前提到的I/O调度有根本的不同。

CFQ I/O调度 算法中，每个进程都有自己的I/O队列，

CFQ I/O调度程序以时间片轮转调度队列，从每个队列中选取一定的请求数(默认4个)，然后进行下一轮调度。

CFQ I/O调度在进程级提供了公平，它的实现位于: block/cfq-iosched.c

#### 3.5 空操作的I/O调度

空操作(noop)I/O调度几乎不做什么事情，这也是它这样命名的原因。

空操作I/O调度只做一件事情，当有新的请求到来时，把它与任一相邻的请求合并。

空操作I/O调度主要用于闪存卡之类的块设备，这类设备没有磁头，没有寻址的负担。

空操作I/O调度的实现位于: block/noop-iosched.c

#### 3.6 I/O调度程序的选择

2.6内核中内置了上面4种I/O调度，可以在启动时通过命令行选项 elevator=xxx 来启用任何一种。

elevator选项参数如下:

|  |  |
| --- | --- |
| **参数** | **I/O调度程序** |
| as | 预测 |
| cfq | 完全公正排队 |
| deadline | 最终期限 |
| noop | 空操作 |

如果启动预测I/O调度，启动的命令行参数中加上 elevator=as

# [（十五）进程地址空间(kernel 2.6.32.60)](http://www.cnblogs.com/wang_yb/p/3351599.html)

进程地址空间也就是每个进程所使用的内存，内核对进程地址空间的管理，也就是对用户态程序的内存管理。

**主要内容**：

* 地址空间(mm\_struct)
* 虚拟内存区域(VMA)
* 地址空间和页表

### 1. 地址空间(mm\_struct)

地址空间就是每个进程所能访问的内存地址范围。

这个地址范围不是真实的，是虚拟地址的范围，有时甚至会超过实际物理内存的大小。

现代的操作系统中进程都是在保护模式下运行的，地址空间其实是操作系统给进程用的一段连续的虚拟内存空间。

地址空间最终会通过页表映射到物理内存上，因为内核操作的是物理内存。

虽然地址空间的范围很大，但是进程也不一定有权限访问全部的地址空间(一般都是只能访问地址空间中的一些地址区间)，

进程能够访问的那些地址区间也称为 内存区域。

进程如果访问了有效内存区域以外的内容就会报 “段错误” 信息。

内存区域中主要包含以下信息：

* - 代码段(text section)，即可执行文件代码的内存映射
* - 数据段(data section)，即可执行文件的已初始化全局变量的内存映射
* - bss段的零页(页面信息全是0值)，即未初始化全局变量的内存映射
* - 进程用户空间栈的零页内存映射
* - 进程使用的C库或者动态链接库等共享库的代码段，数据段和bss段的内存映射
* - 任何内存映射文件
* - 任何共享内存段
* - 任何匿名内存映射，比如由 malloc() 分配的内存

**注**：**bss**是 block started by symbol 的缩写。

linux中内存相关的概念稍微整理了一下，供参考：

|  |  |
| --- | --- |
| **英文** | **含义** |
| SIZE | 进程映射的内存大小，这不是进程实际使用的内存大小 |
| RSS(Resident set size) | 实际驻留在“内存”中的内存大小，不包含已经交换出去的内存 |
| SHARE | RSS中与其他进程共享的内存大小 |
| VMSIZE | 进程占用的总地址空间，包含没有映射到内存中的页 |
| Private RSS | 仅由进程单独占用的RSS，也就是进程实际占用的内存 |

#### 1.1 mm\_struct介绍

linux中的地址空间是用 mm\_struct 来表示的。

下面对其中一些关键的属性进行了注释，有些属性我也不是很了解......

[复制代码](javascript:void(0);)

struct mm\_struct {

struct vm\_area\_struct \* mmap; /\* [内存区域]链表 \*/

struct rb\_root mm\_rb; /\* [内存区域]红黑树 \*/

struct vm\_area\_struct \* mmap\_cache; /\* 最近一次访问的[内存区域] \*/

unsigned long (\*get\_unmapped\_area) (struct file \*filp,

unsigned long addr, unsigned long len,

unsigned long pgoff, unsigned long flags); /\* 获取指定区间内一个还未映射的地址，出错时返回错误码 \*/

void (\*unmap\_area) (struct mm\_struct \*mm, unsigned long addr); /\* 取消地址 addr 的映射 \*/

unsigned long mmap\_base; /\* 地址空间中可以用来映射的首地址 \*/

unsigned long task\_size; /\* 进程的虚拟地址空间大小 \*/

unsigned long cached\_hole\_size; /\* 如果不空的话，就是 free\_area\_cache 后最大的空洞 \*/

unsigned long free\_area\_cache; /\* 地址空间的第一个空洞 \*/

pgd\_t \* pgd; /\* 页全局目录 \*/

atomic\_t mm\_users; /\* 使用地址空间的用户数 \*/

atomic\_t mm\_count; /\* 实际使用地址空间的计数， (users count as 1) \*/

int map\_count; /\* [内存区域]个数 \*/

struct rw\_semaphore mmap\_sem; /\* 内存区域信号量 \*/

spinlock\_t page\_table\_lock; /\* 页表锁 \*/

struct list\_head mmlist; /\* 所有地址空间形成的链表 \*/

/\* Special counters, in some configurations protected by the

\* page\_table\_lock, in other configurations by being atomic.

\*/

mm\_counter\_t \_file\_rss;

mm\_counter\_t \_anon\_rss;

unsigned long hiwater\_rss; /\* High-watermark of RSS usage \*/

unsigned long hiwater\_vm; /\* High-water virtual memory usage \*/

unsigned long total\_vm, locked\_vm, shared\_vm, exec\_vm;

unsigned long stack\_vm, reserved\_vm, def\_flags, nr\_ptes;

unsigned long start\_code, end\_code, start\_data, end\_data; /\* 代码段，数据段的开始和结束地址 \*/

unsigned long start\_brk, brk, start\_stack; /\* 堆的首地址，尾地址，进程栈首地址 \*/

unsigned long arg\_start, arg\_end, env\_start, env\_end; /\* 命令行参数，环境变量首地址，尾地址 \*/

unsigned long saved\_auxv[AT\_VECTOR\_SIZE]; /\* for /proc/PID/auxv \*/

struct linux\_binfmt \*binfmt;

cpumask\_t cpu\_vm\_mask;

/\* Architecture-specific MM context \*/

mm\_context\_t context;

/\* Swap token stuff \*/

/\*

\* Last value of global fault stamp as seen by this process.

\* In other words, this value gives an indication of how long

\* it has been since this task got the token.

\* Look at mm/thrash.c

\*/

unsigned int faultstamp;

unsigned int token\_priority;

unsigned int last\_interval;

unsigned long flags; /\* Must use atomic bitops to access the bits \*/

struct core\_state \*core\_state; /\* coredumping support \*/

#ifdef CONFIG\_AIO

spinlock\_t ioctx\_lock;

struct hlist\_head ioctx\_list;

#endif

#ifdef CONFIG\_MM\_OWNER

/\*

\* "owner" points to a task that is regarded as the canonical

\* user/owner of this mm. All of the following must be true in

\* order for it to be changed:

\*

\* current == mm->owner

\* current->mm != mm

\* new\_owner->mm == mm

\* new\_owner->alloc\_lock is held

\*/

struct task\_struct \*owner;

#endif

#ifdef CONFIG\_PROC\_FS

/\* store ref to file /proc/<pid>/exe symlink points to \*/

struct file \*exe\_file;

unsigned long num\_exe\_file\_vmas;

#endif

#ifdef CONFIG\_MMU\_NOTIFIER

struct mmu\_notifier\_mm \*mmu\_notifier\_mm;

#endif

};

[复制代码](javascript:void(0);)

**补充说明1：** 上面的属性中，mm\_users 和 mm\_count 很容易混淆，这里特别说明一下：(下面的内容有网上查找的，也有我自己理解的)

mm\_users 比较好理解，就是 mm\_struct 被用户空间进程(线程)引用的次数。

如果进程A中创建了3个新线程，那么 进程A(这时候叫线程A也可以)对应的 mm\_struct 中的 mm\_users = 4

补充一点，linux中进程和线程几乎没有什么区别，就是看它是否共享进程地址空间，共享进程地址空间就是线程，反之就是进程。

所以，如果子进程和父进程共享了进程地址空间，那么父子进程都可以看做线程。如果父子进程没有共享进程地址空间，就是2个进程

mm\_count 则稍微有点绕人，其实它记录就是 mm\_struct 实际的引用计数。

简单点说，当 mm\_users=0 时，并不一定能释放此 mm\_struct，只有当 mm\_count=0 时，才可以确定释放此 mm\_struct

从上面的解释可以看出，可能引用 mm\_struct 的并不只是用户空间的进程(线程)

当 mm\_users>0 时， mm\_count 会增加1, 表示有用户空间进程(线程)在使用 mm\_struct。不管使用 mm\_struct 的用户进程(线程)有几个， mm\_count 都只是增加1。

也就是说，如果只有1个进程使用 mm\_struct，那么 mm\_users=1，mm\_count也是 1。

如果有9个线程在使用 mm\_struct，那么 mm\_users=9，而 mm\_count 仍然为 1。

那么 mm\_count 什么情况下会大于 1呢？

当有内核线程使用 mm\_struct 时，mm\_count 才会再增加 1。

内核线程为何会使用用户空间的 mm\_struct 是有其他原因的，这个后面再阐述。这里先知道内核线程使用 mm\_struct 时也会导致 mm\_count 增加 1。

在下面这种情况下，mm\_count 就很有必要了：

* - 进程A启动，并申请了一个 mm\_struct，此时 mm\_users=1, mm\_count=1
* - 进程A中新建了2个线程，此时 mm\_users=3, mm\_count=1
* - 内核调度发生，进程A及相关线程都被挂起，一个内核线程B 使用了进程A 申请的 mm\_struct，此时 mm\_users=3, mm\_count=2
* - CPU的另一个core调度了进程A及其线程，并且执行完了进程A及其线程的所有操作，也就是进程A退出了。此时 mm\_users=0, mm\_count=1
* 在这里就看出 mm\_count 的用处了，如果只有 mm\_users 的话，这里 mm\_users=0 就会释放 mm\_struct，从而有可能导致 内核线程B 异常。
* - 内核线程B 执行完成后退出，这时 mm\_users=0，mm\_count=0，可以安全释放 mm\_struct 了

**补充说明2**：为何内核线程会使用用户空间的 mm\_struct？

对Linux来说，用户进程和内核线程都是task\_struct的实例，

唯一的区别是内核线程是没有进程地址空间的（内核线程使用的内核地址空间），内核线程的mm描述符是NULL，即内核线程的tsk->mm域是空（NULL）。

内核调度程序在进程上下文的时候，会根据tsk->mm判断即将调度的进程是用户进程还是内核线程。

但是虽然内核线程不用访问用户进程地址空间，但是仍然需要页表来访问内核自己的空间。

而任何用户进程来说，他们的内核空间都是100%相同的，所以内核会借用上一个被调用的用户进程的mm\_struct中的页表来访问内核地址，这个mm\_struct就记录在active\_mm。

简而言之就是，对于内核线程，tsk->mm == NULL表示自己内核线程的身份，而tsk->active\_mm是借用上一个用户进程的mm\_struct，用mm\_struct的页表来访问内核空间。

对于用户进程，tsk->mm == tsk->active\_mm。

**补充说明3**：除了 mm\_users 和 mm\_count 之外，还有 mmap 和 mm\_rb 需要说明以下：

其实 mmap 和 mm\_rb 都是保存此 进程地址空间中所有的内存区域(VMA)的，前者是以链表形式存放，后者以红黑树形式存放。

用2种数据结构组织同一种数据是为了便于对VMA进行高效的操作。

#### 1.2 mm\_struct操作

1. 分配进程地址空间

参考 kernel/fork.c 中的宏 allocate\_mm

#define allocate\_mm() (kmem\_cache\_alloc(mm\_cachep, GFP\_KERNEL))

#define free\_mm(mm) (kmem\_cache\_free(mm\_cachep, (mm)))

其实分配进程地址空间时，都是从slab高速缓存中分配的，可以通过 /proc/slabinfo 查看 mm\_struct 的高速缓存

# cat /proc/slabinfo | grep mm\_struct

mm\_struct 35 45 1408 5 2 : tunables 24 12 8 : slabdata 9 9 0

2. 撤销进程地址空间

参考 kernel/exit.c 中的 exit\_mm() 函数

该函数会调用 mmput() 函数减少 mm\_users 的值，

当 mm\_users=0 时，调用 mmdropo() 函数， 减少 mm\_count 的值，

如果 mm\_count=0，那么调用 free\_mm 宏，将 mm\_struct 还给 slab高速缓存

3. 查看进程占用的内存：

cat /proc/<PID>/maps

或者

pmap PID

### 2. 虚拟内存区域(VMA)

内存区域在linux中也被称为虚拟内存区域(VMA)，它其实就是进程地址空间上一段连续的内存范围。

#### 2.1 VMA介绍

VMA的定义也在 <linux/mm\_types.h> 中

[复制代码](javascript:void(0);)

struct vm\_area\_struct {

struct mm\_struct \* vm\_mm; /\* 相关的 mm\_struct 结构体 \*/

unsigned long vm\_start; /\* 内存区域首地址 \*/

unsigned long vm\_end; /\* 内存区域尾地址 \*/

/\* linked list of VM areas per task, sorted by address \*/

struct vm\_area\_struct \*vm\_next, \*vm\_prev; /\* VMA链表 \*/

pgprot\_t vm\_page\_prot; /\* 访问控制权限 \*/

unsigned long vm\_flags; /\* 标志 \*/

struct rb\_node vm\_rb; /\* 树上的VMA节点 \*/

/\*

\* For areas with an address space and backing store,

\* linkage into the address\_space->i\_mmap prio tree, or

\* linkage to the list of like vmas hanging off its node, or

\* linkage of vma in the address\_space->i\_mmap\_nonlinear list.

\*/

union {

struct {

struct list\_head list;

void \*parent; /\* aligns with prio\_tree\_node parent \*/

struct vm\_area\_struct \*head;

} vm\_set;

struct raw\_prio\_tree\_node prio\_tree\_node;

} shared;

/\*

\* A file's MAP\_PRIVATE vma can be in both i\_mmap tree and anon\_vma

\* list, after a COW of one of the file pages. A MAP\_SHARED vma

\* can only be in the i\_mmap tree. An anonymous MAP\_PRIVATE, stack

\* or brk vma (with NULL file) can only be in an anon\_vma list.

\*/

struct list\_head anon\_vma\_node; /\* Serialized by anon\_vma->lock \*/

struct anon\_vma \*anon\_vma; /\* Serialized by page\_table\_lock \*/

/\* Function pointers to deal with this struct. \*/

const struct vm\_operations\_struct \*vm\_ops;

/\* Information about our backing store: \*/

unsigned long vm\_pgoff; /\* Offset (within vm\_file) in PAGE\_SIZE

units, \*not\* PAGE\_CACHE\_SIZE \*/

struct file \* vm\_file; /\* File we map to (can be NULL). \*/

void \* vm\_private\_data; /\* was vm\_pte (shared mem) \*/

unsigned long vm\_truncate\_count;/\* truncate\_count or restart\_addr \*/

#ifndef CONFIG\_MMU

struct vm\_region \*vm\_region; /\* NOMMU mapping region \*/

#endif

#ifdef CONFIG\_NUMA

struct mempolicy \*vm\_policy; /\* NUMA policy for the VMA \*/

#endif

};

[复制代码](javascript:void(0);)

这个结构体各个字段的英文注释都比较详细，就不一一翻译了。

上述属性中的 vm\_flags 标识了此VM 对 VMA和页面的影响：

vm\_flags 的宏定义参见 <linux/mm.h>

|  |  |
| --- | --- |
| **标志** | **对VMA及其页面的影响** |
| VM\_READ | 页面可读取 |
| VM\_WRITE | 页面可写 |
| VM\_EXEC | 页面可执行 |
| VM\_SHARED | 页面可共享 |
| VM\_MAYREAD | VM\_READ 标志可被设置 |
| VM\_MAYWRITER | VM\_WRITE 标志可被设置 |
| VM\_MAYEXEC | VM\_EXEC 标志可被设置 |
| VM\_MAYSHARE | VM\_SHARE 标志可被设置 |
| VM\_GROWSDOWN | 区域可向下增长 |
| VM\_GROWSUP | 区域可向上增长 |
| VM\_SHM | 区域可用作共享内存 |
| VM\_DENYWRITE | 区域映射一个不可写文件 |
| VM\_EXECUTABLE | 区域映射一个可执行文件 |
| VM\_LOCKED | 区域中的页面被锁定 |
| VM\_IO | 区域映射设备I/O空间 |
| VM\_SEQ\_READ | 页面可能会被连续访问 |
| VM\_RAND\_READ | 页面可能会被随机访问 |
| VM\_DONTCOPY | 区域不能在 fork() 时被拷贝 |
| VM\_DONTEXPAND | 区域不能通过 mremap() 增加 |
| VM\_RESERVED | 区域不能被换出 |
| VM\_ACCOUNT | 该区域时一个记账 VM 对象 |
| VM\_HUGETLB | 区域使用了 hugetlb 页面 |
| VM\_NONLINEAR | 该区域是非线性映射的 |

#### 2.2 VMA操作

vm\_area\_struct 结构体定义中有个 vm\_ops 属性，其中定义了内核操作 VMA 的方法

[复制代码](javascript:void(0);)

/\*

\* These are the virtual MM functions - opening of an area, closing and

\* unmapping it (needed to keep files on disk up-to-date etc), pointer

\* to the functions called when a no-page or a wp-page exception occurs.

\*/

struct vm\_operations\_struct {

void (\*open)(struct vm\_area\_struct \* area); /\* 指定内存区域加入到一个地址空间时，该函数被调用 \*/

void (\*close)(struct vm\_area\_struct \* area); /\* 指定内存区域从一个地址空间删除时，该函数被调用 \*/

int (\*fault)(struct vm\_area\_struct \*vma, struct vm\_fault \*vmf); /\* 当没有出现在物理页面中的内存被访问时，该函数被调用 \*/

/\* 当一个之前只读的页面变为可写时，该函数被调用，

\* 如果此函数出错，将导致一个 SIGBUS 信号 \*/

int (\*page\_mkwrite)(struct vm\_area\_struct \*vma, struct vm\_fault \*vmf);

/\* 当 get\_user\_pages() 调用失败时, 该函数被 access\_process\_vm() 函数调用 \*/

int (\*access)(struct vm\_area\_struct \*vma, unsigned long addr,

void \*buf, int len, int write);

#ifdef CONFIG\_NUMA

/\*

\* set\_policy() op must add a reference to any non-NULL @new mempolicy

\* to hold the policy upon return. Caller should pass NULL @new to

\* remove a policy and fall back to surrounding context--i.e. do not

\* install a MPOL\_DEFAULT policy, nor the task or system default

\* mempolicy.

\*/

int (\*set\_policy)(struct vm\_area\_struct \*vma, struct mempolicy \*new);

/\*

\* get\_policy() op must add reference [mpol\_get()] to any policy at

\* (vma,addr) marked as MPOL\_SHARED. The shared policy infrastructure

\* in mm/mempolicy.c will do this automatically.

\* get\_policy() must NOT add a ref if the policy at (vma,addr) is not

\* marked as MPOL\_SHARED. vma policies are protected by the mmap\_sem.

\* If no [shared/vma] mempolicy exists at the addr, get\_policy() op

\* must return NULL--i.e., do not "fallback" to task or system default

\* policy.

\*/

struct mempolicy \*(\*get\_policy)(struct vm\_area\_struct \*vma,

unsigned long addr);

int (\*migrate)(struct vm\_area\_struct \*vma, const nodemask\_t \*from,

const nodemask\_t \*to, unsigned long flags);

#endif

};

[复制代码](javascript:void(0);)

除了以上的操作之外，还有一些辅助函数来方便内核操作内存区域。

这些辅助函数都可以在 <linux/mm.h> 中找到

1. 查找地址空间

[复制代码](javascript:void(0);)

/\* Look up the first VMA which satisfies addr < vm\_end, NULL if none. \*/

extern struct vm\_area\_struct \* find\_vma(struct mm\_struct \* mm, unsigned long addr);

extern struct vm\_area\_struct \* find\_vma\_prev(struct mm\_struct \* mm, unsigned long addr,

struct vm\_area\_struct \*\*pprev);

/\* Look up the first VMA which intersects the interval start\_addr..end\_addr-1,

NULL if none. Assume start\_addr < end\_addr. \*/

static inline struct vm\_area\_struct \* find\_vma\_intersection(struct mm\_struct \* mm, unsigned long start\_addr, unsigned long end\_addr)

{

struct vm\_area\_struct \* vma = find\_vma(mm,start\_addr);

if (vma && end\_addr <= vma->vm\_start)

vma = NULL;

return vma;

}

[复制代码](javascript:void(0);)

2. 创建地址区间

[复制代码](javascript:void(0);)

static inline unsigned long do\_mmap(struct file \*file, unsigned long addr,

unsigned long len, unsigned long prot,

unsigned long flag, unsigned long offset)

{

unsigned long ret = -EINVAL;

if ((offset + PAGE\_ALIGN(len)) < offset)

goto out;

if (!(offset & ~PAGE\_MASK))

ret = do\_mmap\_pgoff(file, addr, len, prot, flag, offset >> PAGE\_SHIFT);

out:

return ret;

}

[复制代码](javascript:void(0);)

3. 删除地址区间

extern int do\_munmap(struct mm\_struct \*, unsigned long, size\_t);

### 3. 地址空间和页表

地址空间中的地址都是虚拟内存中的地址，而CPU需要操作的是物理内存，所以需要一个将虚拟地址映射到物理地址的机制。

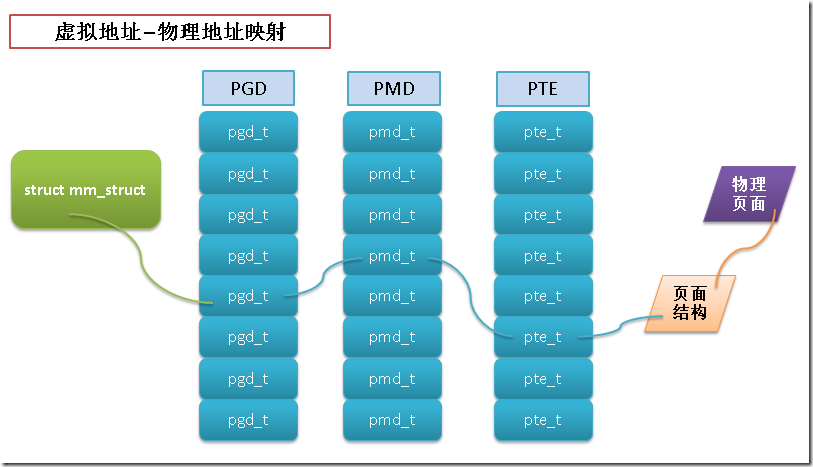
这个机制就是页表，linux中使用3级页面来完成虚拟地址到物理地址的转换。

1. PGD - 全局页目录，包含一个 pgd\_t 类型数组，多数体系结构中 pgd\_t 类型就是一个无符号长整型

2. PMD - 中间页目录，它是个 pmd\_t 类型数组

3. PTE - 简称页表，包含一个 pte\_t 类型的页表项，该页表项指向物理页面

虚拟地址 - 页表 - 物理地址的关系如下图：

[[](https://images0.cnblogs.com/blog/83005/201310/04162809-8efa30e1f1644caf94b1650ad9b9a5b0.png)](https://images0.cnblogs.com/blog/83005/201310/04162809-8efa30e1f1644caf94b1650ad9b9a5b0.png)

# [（十六）- 页高速缓存和页回写](http://www.cnblogs.com/wang_yb/p/3436126.html)

**主要内容：**

* 缓存简介
* 页高速缓存
* 页回写

### ****1. 缓存简介****

在编程中，缓存是很常见也很有效的一种提高程序性能的机制。

linux内核也不例外，为了提高I/O性能，也引入了缓存机制，即将一部分磁盘上的数据缓存到内存中。

#### 1.1 原理

之所以通过缓存能提高I/O性能是基于以下2个重要的原理：

1. CPU访问内存的速度远远大于访问磁盘的速度（访问速度差距不是一般的大，差好几个数量级）
2. 数据一旦被访问，就有可能在短期内再次被访问（临时局部原理）

#### 1.2 策略

缓存的创建和读取没什么好说的，无非就是检查缓存是否存在要创建或者要读取的内容。

但是写缓存和缓存回收就需要好好考虑了，这里面涉及到「缓存内容」和「磁盘内容」同步的问题。

**1.2.1 「写缓存」常见的有3种策略**

* 不缓存(nowrite) :: 也就是不缓存写操作，当对缓存中的数据进行写操作时，直接写入磁盘，同时使此数据的缓存失效
* 写透缓存(write-through) :: 写数据时同时更新磁盘和缓存
* 回写(copy-write or write-behind) :: 写数据时直接写到缓存，由另外的进程(回写进程)在合适的时候将数据同步到磁盘

3种策略的优缺点如下：

|  |  |  |
| --- | --- | --- |
| **策略** | **复杂度** | **性能** |
| 不缓存 | 简单 | 缓存只用于读，对于写操作较多的I/O，性能反而会下降 |
| 写透缓存 | 简单 | 提升了读性能，写性能反而有些下降(除了写磁盘，还要写缓存) |
| 回写 | 复杂 | 读写的性能都有提高(**目前内核中采用的方法**) |

**1.2.2 「缓存回收」的策略**

* 最近最少使用(LRU) :: 每个缓存数据都有个时间戳，保存最近被访问的时间。回收缓存时首先回收时间戳较旧的数据。
* 双链策略(LRU/2) :: 基于LRU的改善策略。具体参见下面的补充说明

**补充说明(双链策略):**

双链策略其实就是 LRU(Least Recently Used) 算法的改进版。

它通过2个链表(活跃链表和非活跃链表)来模拟LRU的过程，目的是为了提高页面回收的性能。

页面回收动作发生时，从非活跃链表的尾部开始回收页面。

双链策略的关键就是页面如何在2个链表之间移动的。

双链策略中，每个页面都有2个标志位，分别为

PG\_active - 标志页面是否活跃，也就是表示此页面是否要移动到活跃链表

PG\_referenced - 表示页面是否被进程访问到

页面移动的流程如下：

1. 当页面第一次被被访问时，PG\_active 置为1，加入到活动链表
2. 当页面再次被访问时，PG\_referenced 置为1，此时如果页面在非活动链表，则将其移动到活动链表，并将PG\_active置为1，PG\_referenced 置为0
3. 系统中 daemon 会定时扫描活动链表，定时将页面的 PG\_referenced 位置为0
4. 系统中 daemon 定时检查页面的 PG\_referenced，如果 PG\_referenced=0，那么将此页面的 PG\_active 置为0，同时将页面移动到非活动链表

参考：[Linux 2.6 中的页面回收与反向映射](http://www.ibm.com/developerworks/cn/linux/l-cn-pagerecycle/index.html)

### ****2. 页高速缓存****

故名思义，页高速缓存中缓存的最小单元就是内存页。

但是此内存页对应的数据不仅仅是文件系统的数据，可以是任何基于页的对象，包括各种类型的文件和内存映射。

#### 2.1 简介

页高速缓存缓存的是具体的物理页面，与前面章节中提到的虚拟内存空间(vm\_area\_struct)不同，假设有进程创建了多个 vm\_area\_struct 都指向同一个文件，

那么这个 vm\_area\_struct 对应的 页高速缓存只有一份。

也就是磁盘上的文件缓存到内存后，它的虚拟内存地址可以有多个，但是物理内存地址却只能有一个。

为了有效提高I/O性能，页高速缓存要需要满足以下条件：

1. 能够快速检索需要的内存页是否存在
2. 能够快速定位 脏页面(也就是被写过，但还没有同步到磁盘上的数据)
3. 页高速缓存被并发访问时，尽量减少并发锁带来的性能损失

下面通过分析内核中的相应的结构体，来了解内核是如何提高 I/O性能的。

#### 2.2 实现

实现页高速缓存的最重要的结构体要算是 address\_space ，在 <linux/fs.h> 中

[复制代码](javascript:void(0);)

struct address\_space {

struct inode \*host; /\* 拥有此 address\_space 的inode对象 \*/

struct radix\_tree\_root page\_tree; /\* 包含全部页面的 radix 树 \*/

spinlock\_t tree\_lock; /\* 保护 radix 树的自旋锁 \*/

unsigned int i\_mmap\_writable;/\* VM\_SHARED 计数 \*/

struct prio\_tree\_root i\_mmap; /\* 私有映射链表的树 \*/

struct list\_head i\_mmap\_nonlinear;/\* VM\_NONLINEAR 链表 \*/

spinlock\_t i\_mmap\_lock; /\* 保护 i\_map 的自旋锁 \*/

unsigned int truncate\_count; /\* 截断计数 \*/

unsigned long nrpages; /\* 总页数 \*/

pgoff\_t writeback\_index;/\* 回写的起始偏移 \*/

const struct address\_space\_operations \*a\_ops; /\* address\_space 的操作表 \*/

unsigned long flags; /\* gfp\_mask 掩码与错误标识 \*/

struct backing\_dev\_info \*backing\_dev\_info; /\* 预读信息 \*/

spinlock\_t private\_lock; /\* 私有 address\_space 自旋锁 \*/

struct list\_head private\_list; /\* 私有 address\_space 链表 \*/

struct address\_space \*assoc\_mapping; /\* 缓冲 \*/

struct mutex unmap\_mutex; /\* 保护未映射页的 mutux 锁 \*/

} \_\_attribute\_\_((aligned(sizeof(long))));

[复制代码](javascript:void(0);)

**补充说明：**

1. inode - 如果 address\_space 是由不带inode的文件系统中的文件映射的话，此字段为 null
2. page\_tree - 这个树结构很重要，它保证了页高速缓存中数据能被快速检索到，脏页面能够快速定位。
3. i\_mmap - 根据 vm\_area\_struct，能够快速的找到关联的缓存文件(即 address\_space)，前面提到过， address\_space 和 vm\_area\_struct 是 一对多的关系。
4. 其他字段主要是提供各种锁和辅助功能

此外，对于这里出现的一种新的数据结构 radix 树，进行简要的说明。

radix树通过long型的位操作来查询各个节点， 存储效率高，并且可以快速查询。

linux中 radix树相关的内容参见： include/linux/radix-tree.h 和 lib/radix-tree.c

下面根据我自己的理解，简单的说明一下radix树结构及原理。

**2.2.1 首先是 radix树节点的定义**

[复制代码](javascript:void(0);)

/\* 源码参照 lib/radix-tree.c \*/

struct radix\_tree\_node {

unsigned int height; /\* radix树的高度 \*/

unsigned int count; /\* 当前节点的子节点数目 \*/

struct rcu\_head rcu\_head; /\* RCU 回调函数链表 \*/

void \*slots[RADIX\_TREE\_MAP\_SIZE]; /\* 节点中的slot数组 \*/

unsigned long tags[RADIX\_TREE\_MAX\_TAGS][RADIX\_TREE\_TAG\_LONGS]; /\* slot标签 \*/

};

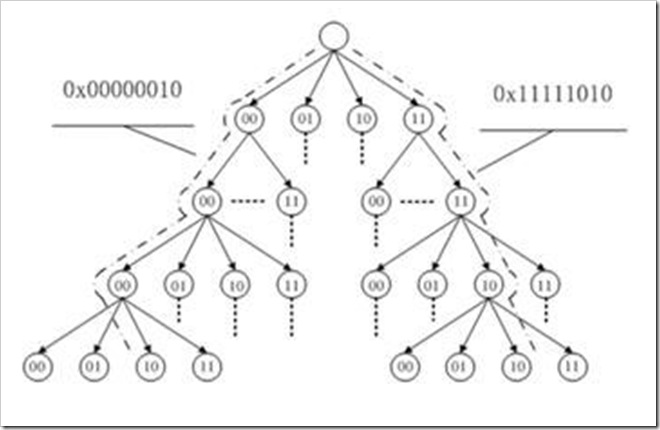
[复制代码](javascript:void(0);)

弄清楚 radix\_tree\_node 中各个字段的含义，也就差不多知道 radix树是怎么一回事了。

* height   表示的整个 radix树的高度（即叶子节点到树根的高度）， 不是当前节点到树根的高度
* count    这个比较好理解，表示当前节点的子节点个数，叶子节点的 count=0
* rcu\_head RCU发生时触发的回调函数链表
* slots    每个slot对应一个子节点（叶子节点）
* tags     标记子节点是否 dirty 或者 wirteback

**2.2.2 每个叶子节点指向文件内相应偏移所对应的缓存页**

比如下图表示 0x000000 至 0x11111111 的偏移范围，树的高度为4 （图是网上找的，不是自己画的）

[](https://images0.cnblogs.com/blog/83005/201311/21181733-d33ba205a792414cba74f5ee38160b61.jpg)

**2.2.3 radix tree 的叶子节点都对应一个二进制的整数，不是字符串，所以进行比较的时候非常快**

其实叶子节点的值就是地址空间的值(一般是long型)

### ****3. 页回写****

由于目前linux内核中对于「写缓存」采用的是第3种策略，所以回写的时机就显得非常重要，回写太频繁影响性能，回写太少容易造成数据丢失。

#### 3.1 简介

linux 页高速缓存中的回写是由内核中的一个线程(flusher 线程)来完成的，flusher 线程在以下3种情况发生时，触发回写操作。

1. 当空闲内存低于一个阀值时

    空闲内存不足时，需要释放一部分缓存，由于只有不脏的页面才能被释放，所以要把脏页面都回写到磁盘，使其变成干净的页面。

2. 当脏页在内存中驻留时间超过一个阀值时

   确保脏页面不会无限期的驻留在内存中，从而减少了数据丢失的风险。

3. 当用户进程调用 sync() 和 fsync() 系统调用时

   给用户提供一种强制回写的方法，应对回写要求严格的场景。

页回写中涉及的一些阀值可以在 /proc/sys/vm 中找到

下表中列出的是与 pdflush(flusher 线程的一种实现) 相关的一些阀值

|  |  |
| --- | --- |
| **阀值** | **描述** |
| dirty\_background\_ratio | 占全部内存的百分比，当内存中的空闲页达到这个比例时，pdflush线程开始回写脏页 |
| dirty\_expire\_interval | 该数值以百分之一秒为单位，它描述超时多久的数据将被周期性执行的pdflush线程写出 |
| dirty\_ratio | 占全部内存的百分比，当一个进程产生的脏页达到这个比例时，就开始被写出 |
| dirty\_writeback\_interval | 该数值以百分之一秒未单位，它描述pdflush线程的运行频率 |
| laptop\_mode | 一个布尔值，用于控制膝上型计算机模式 |

#### 3.2 实现

flusher线程的实现方法随着内核的发展也在不断的变化着。下面介绍几种在内核发展中出现的比较典型的实现方法。

1. 膝上型计算机模式

这种模式的意图是将硬盘转动的机械行为最小化，允许硬盘尽可能长时间的停滞，以此延长电池供电时间。

该模式通过 /proc/sys/vm/laptop\_mode 文件来设置。(0 - 关闭该模式  1 - 开启该模式)

2. bdflush 和 kupdated (2.6版本前 flusher 线程的实现方法)

bdflush 内核线程在后台运行，系统中只有一个 bdflush 线程，当内存消耗到特定阀值以下时，bdflush 线程被唤醒

kupdated 周期性的运行，写回脏页。

**bdflush 存在的问题：**

整个系统仅仅只有一个 bdflush 线程，当系统回写任务较重时，bdflush 线程可能会阻塞在某个磁盘的I/O上，

导致其他磁盘的I/O回写操作不能及时执行。

3. pdflush (2.6版本引入)

pdflush 线程数目是动态的，取决于系统的I/O负载。它是面向系统中所有磁盘的全局任务的。

**pdflush 存在的问题：**

pdflush的数目是动态的，一定程度上缓解了 bdflush 的问题。但是由于 pdflush 是面向所有磁盘的，

所以有可能出现多个 pdflush 线程全部阻塞在某个拥塞的磁盘上，同样导致其他磁盘的I/O回写不能及时执行。

4. flusher线程 (2.6.32版本后引入)

flusher线程改善了上面出现的问题：

首先，flusher 线程的数目不是唯一的，这就避免了 bdflush 线程的问题

其次，flusher 线程不是面向所有磁盘的，而是每个 flusher 线程对应一个磁盘，这就避免了 pdflush 线程的问题

# [（十七）- 设备与模块](http://www.cnblogs.com/wang_yb/p/3489753.html)

本章主要讨论与linux的设备驱动和设备管理的相关的4个内核成分，设备类型，模块，内核对象，sysfs。

**主要内容：**

* 设备类型
* 内核模块
* 内核对象
* sysfs
* 总结

### ****1. 设备类型****

linux中主要由3种类型的设备，分别是：

|  |  |  |  |
| --- | --- | --- | --- |
| **设备类型** | **代表设备** | **特点** | **访问方式** |
| 块设备 | 硬盘，光盘 | 随机访问设备中的内容 | 一般都是把设备挂载为文件系统后再访问 |
| 字符设备 | 键盘，打印机 | 只能顺序访问(一个一个字符或者一个一个字节) | 一般不挂载，直接和设备交互 |
| 网络设备 | 网卡 | 打破了Unix "所有东西都是文件" 的设计原则 | 通过套接字API来访问 |

除了以上3种典型的设备之外，其实Linux中还有一些其他的设备类型，其中见的较多的应该算是"伪设备"。

所谓"伪设备"，其实就是一些虚拟的设备，仅提供访问内核功能而已，没有物理设备与之关联。

典型的"伪设备"就是 /dev/random(内核随机数发生器)， /dev/null(空设备)， /dev/zero(零设备)， /dev/full(满设备)

### ****2. 内核模块****

Linux内核是模块化组成的，内核中的模块可以按需加载，从而保证内核启动时不用加载所有的模块，即减少了内核的大小，也提高了效率。

通过编写内核模块来给内核增加功能或者接口是个很好的方式（既不用重新编译内核，也方便调试和删除）。

#### 2.1 内核模块示例

内核模块可以带参数也可以不带参数，不带参数的内核模块比较简单。

我之前的几篇随笔中用于测试的例子都是用不带参数的内核模块来实验的。

**2.1.1. 无参数的内核模块**

参考：

[《Linux内核设计与实现》读书笔记（六）- 内核数据结构](http://www.cnblogs.com/wang_yb/archive/2013/04/16/3023892.html)

[《Linux内核设计与实现》读书笔记（八）- 中断下半部的处理](http://www.cnblogs.com/wang_yb/archive/2013/04/23/3037268.html)

[《Linux内核设计与实现》读书笔记（十一）- 定时器和时间管理](http://www.cnblogs.com/wang_yb/archive/2013/05/10/3070373.html)

**2.1.2. 带参数的内核模块**

构造带参数的内核模块其实也不难，内核中已经提供了简单的框架来给我们声明参数。

1. module\_param(name, type, perm) : 定义一个模块参数

+ 参数 name :: 既是用户可见的参数名，也是模块中存放模块参数的变量名

+ 参数 type :: 参数的类型(byte, short, int, uint, long, ulong, charp, bool...) byte型存放在char变量中，bool型存放在int变量中

+ 参数 perm :: 指定模块在 sysfs 文件系统中对应的文件权限(关于 sysfs 的内容后面介绍)

static int stu\_id = 0; // 默认id

module\_param(stu\_id, int, 0644);

2. module\_param\_named(name, variable, type, perm) : 定义一个模块参数，并且参数对内对外的名称不一样

+ 参数 name :: 用户可见的参数名

+ 参数 variable :: 模块中存放模块参数的变量名

+ 参数 type和perm :: 同 module\_param 中的 type 和 perm

static char\* stu\_name\_in = "default name"; // 默认名字

module\_param\_named(stu\_name\_out, stu\_name\_in ,charp, 0644);

/\* stu\_name\_out 是对用户开放的名称

\* stu\_name\_in 是内核模块内部使用的名称

\*/

3. module\_param\_string(name, string, len, perm) : 拷贝字符串到指定的字符数组

+ 参数 name :: 用户可见的参数名

+ 参数 string :: 模块中存放模块参数的变量名

+ 参数 len :: string 参数的缓冲区长度

+ 参数 perm :: 同 module\_param 中的 perm

static char str\_in[BUF\_LEN];

module\_param\_string(str\_out, str\_in, BUF\_LEN, 0);

/\* perm=0 表示完全禁止 sysfs 项 \*/

4. module\_param\_array(name, type, nump, perm) : 定义数组类型的模块参数

+ 参数 name :: 同 module\_param 中的 name

+ 参数 type :: 同 module\_param 中的 type

+ 参数 nump :: 整型指针，存放数组的长度

+ 参数 perm :: 同 module\_param 中的 perm

#define MAX\_ARR\_LEN 5

static int arr\_len;

static int arr\_in[MAX\_ARR\_LEN];

module\_param\_array(arr\_in, int, &arr\_len, 0644);

5. module\_param\_array\_named(name, array, type, nump, perm) : 定义数组类型的模块参数,并且数组参数对内对外的名称不一样

+ 参数 name :: 数组参数对外的名称

+ 参数 array :: 数组参数对内的名称

+ 参数 type，nump，perm :: 同 module\_param\_array 中的 type，nump，perm

#define MAX\_ARR\_LEN 5

static int arr\_len;

static int arr\_in[MAX\_ARR\_LEN];

module\_param\_array\_named(arr\_out, arr\_in, int, &arr\_len, 0644);

6. 参数描述宏

可以通过 MODULE\_PARM\_DESC() 来给内核模块的参数添加一些描述信息。

这些描述信息在编译完内核模块后，可以通过 modinfo  命令查看。

static int stu\_id = 0; // 默认id

module\_param(stu\_id, int, 0644);

MODULE\_PARM\_DESC(stu\_id, "学生ID，默认为 0"); // 这句就是描述内核模块参数 stu\_id 的语句

7. 带参数的内核模块的示例

示例代码：test\_paramed\_km.c

定义了3个内核模块参数，分别是 int型，char\*型，数组型的。

[复制代码](javascript:void(0);)

#include<linux/init.h>

#include<linux/module.h>

#include<linux/kernel.h>

MODULE\_LICENSE("Dual BSD/GPL");

struct student

{

int id;

char\* name;

};

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

static int stu\_id = 0; // 默认id

module\_param(stu\_id, int, 0644);

MODULE\_PARM\_DESC(stu\_id, "学生ID，默认为 0");

static char\* stu\_name\_in = "default name"; // 默认名字

module\_param\_named(stu\_name\_out, stu\_name\_in ,charp, 0644);

MODULE\_PARM\_DESC(stu\_name, "学生姓名，默认为 default name");

#define MAX\_ARR\_LEN 5

static int arr\_len;

static int arr\_in[MAX\_ARR\_LEN];

module\_param\_array\_named(arr\_out, arr\_in, int, &arr\_len, 0644);

MODULE\_PARM\_DESC(arr\_in, "数组参数，默认为空");

static int test\_paramed\_km\_init(void)

{

struct student\* stu1;

int i;

/\* 进入内核模块 \*/

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

printk(KERN\_ALERT "test\_paramed\_km is inited!\n");

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

// 根据参数生成 struct student 信息

// 如果没有参数就用默认参数

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

stu1 = kmalloc(sizeof(\*stu1), GFP\_KERNEL);

stu1->id = stu\_id;

stu1->name = stu\_name\_in;

print\_student(stu1);

// 模块数组

for (i = 0; i < arr\_len; ++i) {

printk(KERN\_ALERT "arr\_value[%d]: %d\n", i, arr\_in[i]);

}

return 0;

}

static void test\_paramed\_km\_exit(void)

{

/\* 退出内核模块 \*/

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

printk(KERN\_ALERT "test\_paramed\_km is exited!\n");

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

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

}

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(test\_paramed\_km\_init);

module\_exit(test\_paramed\_km\_exit);

[复制代码](javascript:void(0);)

上面的示例对应的 Makefile 如下：

[复制代码](javascript:void(0);)

# must complile on customize kernel

obj-m += paramed\_km.o

paramed\_km-objs := test\_paramed\_km.o

#generate the path

CURRENT\_PATH:=$(shell pwd)

#the current kernel version number

LINUX\_KERNEL:=$(shell uname -r)

#the absolute path

LINUX\_KERNEL\_PATH:=/usr/src/kernels/$(LINUX\_KERNEL)

#complie object

all:

make -C $(LINUX\_KERNEL\_PATH) M=$(CURRENT\_PATH) modules

rm -rf modules.order Module.symvers .\*.cmd \*.o \*.mod.c .tmp\_versions \*.unsigned

#clean

clean:

rm -rf modules.order Module.symvers .\*.cmd \*.o \*.mod.c \*.ko .tmp\_versions \*.unsigned

[复制代码](javascript:void(0);)

内核模块运行方法：(我的运行环境是 CentOS 6.3 x86\_64)

[复制代码](javascript:void(0);)

[root@vbox chap17]# uname -r

2.6.32-279.el6.x86\_64

[root@vbox chap17]# ll

total 8

-rw-r--r-- 1 root root 538 Dec 1 19:37 Makefile

-rw-r--r-- 1 root root 2155 Dec 1 19:37 test\_paramed\_km.c

[root@vbox chap17]# make <-- 编译内核

make -C /usr/src/kernels/2.6.32-279.el6.x86\_64 M=/root/chap17 modules

make[1]: Entering directory `/usr/src/kernels/2.6.32-279.el6.x86\_64'

CC [M] /root/chap17/test\_paramed\_km.o

LD [M] /root/chap17/paramed\_km.o

Building modules, stage 2.

MODPOST 1 modules

CC /root/chap17/paramed\_km.mod.o

LD [M] /root/chap17/paramed\_km.ko.unsigned

NO SIGN [M] /root/chap17/paramed\_km.ko

make[1]: Leaving directory `/usr/src/kernels/2.6.32-279.el6.x86\_64'

rm -rf modules.order Module.symvers .\*.cmd \*.o \*.mod.c .tmp\_versions \*.unsigned

[root@vbox chap17]# ll <-- 编译内核后，多了 paramed\_km.ko 文件

total 124

-rw-r--r-- 1 root root 538 Dec 1 19:37 Makefile

-rw-r--r-- 1 root root 118352 Dec 1 19:37 paramed\_km.ko

-rw-r--r-- 1 root root 2155 Dec 1 19:37 test\_paramed\_km.c

<-- 通过 modinfo 命令可以查看对内核模块参数的注释

[root@vbox chap17]# modinfo paramed\_km.ko

filename: paramed\_km.ko

license: Dual BSD/GPL

srcversion: C52F97687B033738742800D

depends:

vermagic: 2.6.32-279.el6.x86\_64 SMP mod\_unload modversions

parm: stu\_id:学生ID，默认为 0 (int)

parm: stu\_name\_out:charp

parm: stu\_name\_in:学生姓名，默认为 default name

parm: arr\_out:array of int

parm: arr\_in:数组参数，默认为空

<-- 3 个参数都是默认的

[root@vbox chap17]# insmod paramed\_km.ko

[root@vbox chap17]# rmmod paramed\_km.ko

[root@vbox chap17]# dmesg | tail -16 <-- 结果中显示2个默认参数，第3个数组参数默认为空，所以不显示

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

test\_paramed\_km is inited!

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

alloc one student....

\*\*\*\*\*\*\*\*\*\*student info\*\*\*\*\*\*\*\*\*\*\*

student id is: 0

student name is: default name

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

test\_paramed\_km is exited!

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

<-- 3 个参数都被设置

[root@vbox chap17]# insmod paramed\_km.ko stu\_id=100 stu\_name\_out=myname arr\_out=1,2,3,4,5

[root@vbox chap17]# rmmod paramed\_km.ko

[root@vbox chap17]# dmesg | tail -21

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

test\_paramed\_km is inited!

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

alloc one student....

\*\*\*\*\*\*\*\*\*\*student info\*\*\*\*\*\*\*\*\*\*\*

student id is: 100

student name is: myname

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

arr\_value[0]: 1

arr\_value[1]: 2

arr\_value[2]: 3

arr\_value[3]: 4

arr\_value[4]: 5

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

test\_paramed\_km is exited!

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

[复制代码](javascript:void(0);)

#### 2.2 内核模块的位置

2.2.1.  内核代码外

上面的例子，以及之前博客中内核模块的例子都是把模块代码放在内核之外来运行的。

2.2.2. 内核代码中

内核模块的代码也可以直接放到内核代码树中。

如果你开发了一种驱动，并且希望被加入到内核中，那么，可以在编写驱动的时候就将完成此驱动功能的内核模块加到内核代码树中 driver 的相应位置。

将内核模块加入内核代码树中之后，不需要另外写 Makefile，修改内核代码树中的已有的 Makefile 就行。

比如，写了一个某种字符设备相关的驱动，可以把它加到内核代码的 /drivers/char 下，

同时修改 /drivers/char下的Makefie，仿照里面已有的内容，增加新驱动的编译相关内容即可。

之后，在编译内核的时候会将新的驱动以内核模块的方式编译出来。

#### 2.3 内核模块相关操作

2.3.1. 模块安装

make modules\_install <-- 把随内核编译出来的模块安装到合适的目录中( /lib/modules/version/kernel )

2.3.2. 模块依赖性

linux中自动生产模块依赖性的命令：

depmod <-- 产生内核依赖关系信息

depmod -A <-- 只为新模块生成依赖信息(速度更快)

2.3.3. 模块的载入

内核模块实验时已经用过：

insmod module.ko

<-- 推荐使用以下的命令， 自动加载依赖的模块

modprobe module [module parameters]

2.3.4. 模块的卸载

内核模块实验时已经用过：

rmmod module.ko

<-- 推荐使用以下的命令， 自动卸载依赖的模块

modprobe -r module

2.3.5. 模块导出符号表

内核模块被载入后，就动态的加载到内核中，为了能让其他内核模块使用其功能，需要将其中函数导出。

内核模块中导出函数的方法：

EXPORT\_SYMBOL(函数名) <-- 接在要导出的函数后面即可

EXPORT\_SYMBOL\_GPL(函数名) <-- 和EXPORT\_SYMBOL一样，区别在于只对标记为GPL协议的模块可见

内核模块导出符号表 **示例**：

+ 首先编写一个导出函数的模块 module\_A: test\_module\_A.c

[复制代码](javascript:void(0);)

#include<linux/init.h>

#include<linux/module.h>

#include<linux/kernel.h>

MODULE\_LICENSE("Dual BSD/GPL");

static int test\_export\_A\_init(void)

{

/\* 进入内核模块 \*/

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

printk(KERN\_ALERT "ENTRY test\_export\_A!\n");

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

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

return 0;

}

static void test\_export\_A\_exit(void)

{

/\* 退出内核模块 \*/

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

printk(KERN\_ALERT "EXIT test\_export\_A!\n");

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

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

}

/\* 要导出的函数 \*/

int export\_add10(int param)

{

printk(KERN\_ALERT "param from other module is : %d\n", param);

return param + 10;

}

EXPORT\_SYMBOL(export\_add10);

module\_init(test\_export\_A\_init);

module\_exit(test\_export\_A\_exit);

[复制代码](javascript:void(0);)

test\_module\_A.c 的 Makefile

[复制代码](javascript:void(0);)

# must complile on customize kernel

obj-m += export\_A.o

export\_A-objs := test\_export\_A.o

#generate the path

CURRENT\_PATH:=$(shell pwd)

#the current kernel version number

LINUX\_KERNEL:=$(shell uname -r)

#the absolute path

LINUX\_KERNEL\_PATH:=/usr/src/kernels/$(LINUX\_KERNEL)

#complie object

all:

make -C $(LINUX\_KERNEL\_PATH) M=$(CURRENT\_PATH) modules

rm -rf modules.order .\*.cmd \*.o \*.mod.c .tmp\_versions \*.unsigned

#clean

clean:

rm -rf modules.order Module.symvers .\*.cmd \*.o \*.mod.c \*.ko .tmp\_versions \*.unsigned

[复制代码](javascript:void(0);)

+ 再编写一个内核模块 module\_B，使用 module\_A 导出的函数 : test\_module\_B.c

[复制代码](javascript:void(0);)

#include<linux/init.h>

#include<linux/module.h>

#include<linux/kernel.h>

MODULE\_LICENSE("Dual BSD/GPL");

extern int export\_add10(int); // 这个函数是 module\_A 中实现的

static int test\_export\_B\_init(void)

{

/\* 进入内核模块 \*/

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

printk(KERN\_ALERT "ENTRY test\_export\_B!\n");

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

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

/\* 调用 module\_A 导出的函数 \*/

printk(KERN\_ALERT "result from test\_export\_A: %d\n", export\_add10(100));

return 0;

}

static void test\_export\_B\_exit(void)

{

/\* 退出内核模块 \*/

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

printk(KERN\_ALERT "EXIT test\_export\_B!\n");

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

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

}

module\_init(test\_export\_B\_init);

module\_exit(test\_export\_B\_exit);

[复制代码](javascript:void(0);)

test\_module\_B.c 的 Makefile

[复制代码](javascript:void(0);)

# must complile on customize kernel

obj-m += export\_B.o

export\_B-objs := test\_export\_B.o

#generate the path

CURRENT\_PATH:=$(shell pwd)

#the current kernel version number

LINUX\_KERNEL:=$(shell uname -r)

#the absolute path

LINUX\_KERNEL\_PATH:=/usr/src/kernels/$(LINUX\_KERNEL)

#complie object

all:

make -C $(LINUX\_KERNEL\_PATH) M=$(CURRENT\_PATH) modules

rm -rf modules.order Module.symvers .\*.cmd \*.o \*.mod.c .tmp\_versions \*.unsigned

#clean

clean:

rm -rf modules.order Module.symvers .\*.cmd \*.o \*.mod.c \*.ko .tmp\_versions \*.unsigned

[复制代码](javascript:void(0);)

+ 测试方法

1. 将 test\_export\_A.c 和对应的 Makefile 拷贝到 module\_A 文件夹中

2. 将 test\_export\_B.c 和对应的 Makefile 拷贝到 module\_B 文件夹中

3. 编译 module\_A 中的 test\_export\_A.c

4. 将编译 module\_A 后生成的 Module.symvers 拷贝到 module\_B 文件夹中

5. 编译 module\_B 中的 test\_export\_B.c

6. 先安装 模块A，再安装模块B

7. dmesg 查看log

8. 用 rmmod 卸载模块B 和 模块A (注意卸载顺序，先卸载B再卸载A)

[复制代码](javascript:void(0);)

[root@vbox chap17]# ll

total 8

drwxrwxr-x 2 root root 4096 Dec 7 22:14 module\_A

drwxrwxr-x 2 root root 4096 Dec 7 22:14 module\_B

[root@vbox chap17]# ll module\_A

total 8

-rw-r--r-- 1 root root 517 Dec 7 21:58 Makefile

-rw-r--r-- 1 root root 893 Dec 7 21:58 test\_export\_A.c

[root@vbox chap17]# ll module\_B

total 8

-rw-r--r-- 1 root root 532 Dec 7 21:58 Makefile

-rw-r--r-- 1 root root 830 Dec 7 21:58 test\_export\_B.c

[root@vbox chap17]# cd module\_A/

[root@vbox module\_A]# ll

total 8

-rw-r--r-- 1 root root 517 Dec 7 21:58 Makefile

-rw-r--r-- 1 root root 893 Dec 7 21:58 test\_export\_A.c

[root@vbox module\_A]# make

make -C /usr/src/kernels/2.6.32-279.el6.x86\_64 M=/root/chap17/module\_A modules

make[1]: Entering directory `/usr/src/kernels/2.6.32-279.el6.x86\_64'

CC [M] /root/chap17/module\_A/test\_export\_A.o

LD [M] /root/chap17/module\_A/export\_A.o

Building modules, stage 2.

MODPOST 1 modules

CC /root/chap17/module\_A/export\_A.mod.o

LD [M] /root/chap17/module\_A/export\_A.ko.unsigned

NO SIGN [M] /root/chap17/module\_A/export\_A.ko

make[1]: Leaving directory `/usr/src/kernels/2.6.32-279.el6.x86\_64'

rm -rf modules.order .\*.cmd \*.o \*.mod.c .tmp\_versions \*.unsigned

[root@vbox module\_A]# ll

total 120

-rw-r--r-- 1 root root 110452 Dec 7 22:31 export\_A.ko

-rw-r--r-- 1 root root 517 Dec 7 21:58 Makefile

-rw-r--r-- 1 root root 69 Dec 7 22:31 Module.symvers

-rw-r--r-- 1 root root 893 Dec 7 21:58 test\_export\_A.c

[root@vbox module\_A]# cd ../module\_B

[root@vbox module\_B]# ll

total 8

-rw-r--r-- 1 root root 532 Dec 7 21:58 Makefile

-rw-r--r-- 1 root root 830 Dec 7 21:58 test\_export\_B.c

[root@vbox module\_B]# cp ../module\_A/Module.symvers .

[root@vbox module\_B]# ll

total 12

-rw-r--r-- 1 root root 532 Dec 7 21:58 Makefile

-rw-r--r-- 1 root root 69 Dec 7 22:32 Module.symvers

-rw-r--r-- 1 root root 830 Dec 7 21:58 test\_export\_B.c

[root@vbox module\_B]# make

make -C /usr/src/kernels/2.6.32-279.el6.x86\_64 M=/root/chap17/module\_B modules

make[1]: Entering directory `/usr/src/kernels/2.6.32-279.el6.x86\_64'

CC [M] /root/chap17/module\_B/test\_export\_B.o

LD [M] /root/chap17/module\_B/export\_B.o

Building modules, stage 2.

MODPOST 1 modules

CC /root/chap17/module\_B/export\_B.mod.o

LD [M] /root/chap17/module\_B/export\_B.ko.unsigned

NO SIGN [M] /root/chap17/module\_B/export\_B.ko

make[1]: Leaving directory `/usr/src/kernels/2.6.32-279.el6.x86\_64'

rm -rf modules.order Module.symvers .\*.cmd \*.o \*.mod.c .tmp\_versions \*.unsigned

[root@vbox module\_B]# ll

total 116

-rw-r--r-- 1 root root 108596 Dec 7 22:32 export\_B.ko

-rw-r--r-- 1 root root 532 Dec 7 21:58 Makefile

-rw-r--r-- 1 root root 830 Dec 7 21:58 test\_export\_B.c

[root@vbox module\_B]# insmod ../module\_A/export\_A.ko

[root@vbox module\_B]# insmod export\_B.ko

[root@vbox module\_B]# dmesg | tail -18

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ENTRY test\_export\_A!

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ENTRY test\_export\_B!

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

param from other module is : 100

result from test\_export\_A: 110

[root@vbox module\_B]# rmmod export\_B

[root@vbox module\_B]# rmmod export\_A

[复制代码](javascript:void(0);)

**注：**1. 必须把编译模块A后生成的 Module.symvers 拷贝到module\_B 中再编译模块B，否在模块B找不到模块A导出的函数

2. 先安装模块A，再安装模块B。

3. 先卸载模块B，再卸载模块A。

4. 安装卸载如果不按照上面的顺序，会有错误提示，大家可以试试看。

5. 我实验的系统是 CentOS6.3 x86\_64

### ****3. 内核对象****

2.6内核中增加了一个引人注目的新特性--统一设备模型(device model)。

统一设备模型的最初动机是为了实现智能的电源管理，linux 内核为了实现智能电源管理，需要建立表示系统中所有设备拓扑关系的树结构，

这样在关闭电源时，可以从树的节点开始关闭。

实现了统一设备模型之后，还给内核带来了如下的好处:

1. 代码重复最小化（统一处理的东西多了）

2. 可以列举系统中所有设备，观察它们的状态，并查看它们连接的总线

3. 可以将系统中的全部设备以树的形式完整，有效的展示出来--包括所有总线和内部连接

4. 可以将设备和其对应的驱动联系起来，反之亦然

5. 可以将设备按照类型加以归类，无需理解物理设备的拓扑结构

6. 可以沿设备树的叶子向其根的反向依次遍历，以保证能以正确的顺序关闭设备电源

#### 3.1 kobject 简介

统一设备模型的核心部分就是 kobject，通过下面对kobject结构体的介绍，可以大致了解它是如何使得各个物理设备能够以树结构的形式组织起来的。

3.1.1. kobject

kobject的定义在 <linux/kobject.h> 中

[复制代码](javascript:void(0);)

struct kobject {

const char \*name; /\* kobject 名称 \*/

struct list\_head entry; /\* kobject 链表 \*/

struct kobject \*parent; /\* kobject 的父对象，说明kobject是有层次结构的 \*/

struct kset \*kset; /\* kobject 的集合，接下来有详细介绍 \*/

struct kobj\_type \*ktype; /\* kobject 的类型，接下来有详细介绍 \*/

struct sysfs\_dirent \*sd; /\* 在sysfs中，这个结构体表示kobject的一个inode结构体，sysfs之后也会介绍 \*/

struct kref kref; /\* 提供 kobject 的引用计数 \*/

/\* 一些标志位 \*/

unsigned int state\_initialized:1;

unsigned int state\_in\_sysfs:1;

unsigned int state\_add\_uevent\_sent:1;

unsigned int state\_remove\_uevent\_sent:1;

unsigned int uevent\_suppress:1;

};

[复制代码](javascript:void(0);)

kobject 本身不代表什么实际的内容，一般都是嵌在其他数据结构中来发挥作用。（感觉有点像内核数据结构链表的节点）

比如 <linux/cdev.h> 中的 struct cdev （表示字符设备的struct）

[复制代码](javascript:void(0);)

struct cdev {

struct kobject kobj; /\* 嵌在 cdev 中的kobject \*/

struct module \*owner;

const struct file\_operations \*ops;

struct list\_head list;

dev\_t dev;

unsigned int count;

};

[复制代码](javascript:void(0);)

cdev中嵌入了kobject之后，就可以通过 cdev->kboj.parent 建立cdev之间的层次关系，通过 cdev->kobj.entry 获取关联的所有cdev设备等。

总之，嵌入了kobject之后，cdev设备之间就有了树结构关系，cdev设备和其他设备之间也有可层次关系。

3.1.2. ktype

ktype是为了描述一族的kobject所具有的普遍属性，也就是将这一族的kobject的属性统一定义一下，避免每个kobject分别定义。

（感觉有点像面向对象语言中的抽象类或者接口）

ktype的定义很简单，参见<linux/kobject.h>

struct kobj\_type {

void (\*release)(struct kobject \*kobj); /\* kobject的引用计数降到0时触发的析构函数，负责释放和清理内存的工作 \*/

struct sysfs\_ops \*sysfs\_ops; /\* sysfs操作相关的函数 \*/

struct attribute \*\*default\_attrs; /\* kobject 相关的默认属性 \*/

};

3.1.3. kset

kset是kobject对象的集合体，可以所有相关的kobject置于一个kset之中，比如所有“块设备”可以放在一个表示块设备的kset中。

kset的定义也不复杂，参见 <linux/kobject.h>

struct kset {

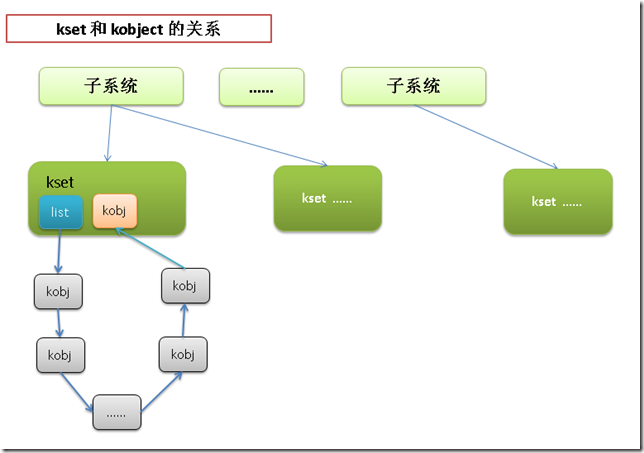
struct list\_head list; /\* 表示kset中所有kobject的链表 \*/

spinlock\_t list\_lock; /\* 用于保护 list 的自旋锁\*/

struct kobject kobj; /\* kset中嵌入的一个kobject，使得kset也可以表现的像一样kobject一样\*/

struct kset\_uevent\_ops \*uevent\_ops; /\* 处理kset中kobject的热插拔事件 提供了与用户空间热插拔进行通信的机制 \*/

};

[](https://images0.cnblogs.com/blog/83005/201312/24215225-7ae8cf55c80c4e8ebecf988b05f1424e.png)

3.1.4. kobject，ktype和kset之间的关系

这3个概念中，kobject是最基本的。kset和ktype是为了将kobject进行分类，以便将共通的处理集中处理，从而减少代码量，也增加维护性。

这里kset和ktype都是为了将kobject进行分类，为什么会有2中分类呢？

从整个内核的代码来看，其实kset的数量是多于ktype的数量的，同一种ktype的kobject可以位于不同的kset中。

做个不是很恰当的比喻，如果把kobject比作一个人的话，kset相当于一个一个国家，ktype则相当于人种(比如黄种人，白种人等等)。

人种的类型只有少数几个，但是国家确有很多，人种的目的是描述一群人的共通属性，而国家的目地则是为了管理一群人。

同样，ktype侧重于描述，kset侧重于管理。

3.1.5. kref

kref记录kobject被引用的次数，当引用计数降到0的时候，则执行release函数释放相关资源。

kref的定义参见：<linux/kref.h>

[复制代码](javascript:void(0);)

struct kref {

atomic\_t refcount; /\* 只有一个表示引用计数的属性，atomic\_t 类型表示对它的访问是原子操作 \*/

};

void kref\_set(struct kref \*kref, int num); /\* 设置引用计数的值 \*/

void kref\_init(struct kref \*kref); /\* 初始化引用计数 \*/

void kref\_get(struct kref \*kref); /\* 增加引用计数 +1 \*/

int kref\_put(struct kref \*kref, void (\*release) (struct kref \*kref)); /\* 减少引用计数 -1 当减少到0时，释放相应资源 \*/

[复制代码](javascript:void(0);)

上面这些函数的具体实现可以参考内核代码  lib/kref.c

#### 3.2 kobject 操作

kobject的相关都在 <linux/kobject.h> 中定义了，主要由以下一些：

[复制代码](javascript:void(0);)

extern void kobject\_init(struct kobject \*kobj, struct kobj\_type \*ktype); /\* 初始化一个kobject，设置它是哪种ktype \*/

extern int \_\_must\_check kobject\_add(struct kobject \*kobj,

struct kobject \*parent,

const char \*fmt, ...); /\* 设置此kobject的parent，将此kobject加入到现有对象层次结构中 \*/

extern int \_\_must\_check kobject\_init\_and\_add(struct kobject \*kobj,

struct kobj\_type \*ktype,

struct kobject \*parent,

const char \*fmt, ...); /\* 初始化kobject，完成kobject\_add 函数的功能\*/

extern void kobject\_del(struct kobject \*kobj); /\* 将此kobject从现有对象层次结构中取消 \*/

extern struct kobject \* \_\_must\_check kobject\_create(void); /\* 创建一个kobject，比kobject\_init更常用 \*/

extern struct kobject \* \_\_must\_check kobject\_create\_and\_add(const char \*name,

struct kobject \*parent); /\* 创建一个kobject，并将其加入到现有对象层次结构中 \*/

extern int \_\_must\_check kobject\_rename(struct kobject \*, const char \*new\_name); /\* 改变kobject的名称 \*/

extern int \_\_must\_check kobject\_move(struct kobject \*, struct kobject \*); /\* 给kobject设置新的parent \*/

extern struct kobject \*kobject\_get(struct kobject \*kobj); /\* 增加kobject的引用计数 +1 \*/

extern void kobject\_put(struct kobject \*kobj); /\* 减少kobject的引用计数 -1 \*/

extern char \*kobject\_get\_path(struct kobject \*kobj, gfp\_t flag); /\* 生成并返回与给定的一个kobj和kset相关联的路径 \*/

[复制代码](javascript:void(0);)

上面这些函数的具体实现可以参考内核代码  lib/kobject.c

### ****4. sysfs****

sysfs是一个处于内存中的虚拟文件系统，它提供了kobject对象层次结构的视图。

可以用下面这个命令来查看 /sys 的结构

tree /sys # 显示所有目录和文件

或者

tree -L 1 /sys # 只显示一层目录

#### 4.1 sysfs中的kobject

既然sysfs是kobject的视图，那么内核中肯定提供了在sysfs中操作kobject的API。

kobject结构体中与sysfs关联的字段就是 「struct sysfs\_dirent    \*sd; 」这是一个目录项结构，它表示kobject在sysfs中的位置。

关于目录项，可以参考：[《Linux内核设计与实现》读书笔记（十三）- 虚拟文件系统](http://www.cnblogs.com/wang_yb/p/3144291.html)

4.1.1. sysfs中添加和删除kobject非常简单，就是上面介绍的 kobject操作中提到的

[复制代码](javascript:void(0);)

extern int \_\_must\_check kobject\_add(struct kobject \*kobj,

struct kobject \*parent,

const char \*fmt, ...); /\* 设置此kobject的parent，将此kobject加入到现有对象层次结构中 \*/

extern int \_\_must\_check kobject\_init\_and\_add(struct kobject \*kobj,

struct kobj\_type \*ktype,

struct kobject \*parent,

const char \*fmt, ...); /\* 初始化kobject，完成kobject\_add 函数的功能\*/

extern void kobject\_del(struct kobject \*kobj); /\* 将此kobject从现有对象层次结构中取消 \*/

... ...等等

[复制代码](javascript:void(0);)

添加了kobject之后，只会增加文件夹，不会增加文件。

因为kobject在sysfs中就是映射成一个文件夹。

添加删除kobject的示例代码如下：

[复制代码](javascript:void(0);)

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* @file : test\_kobject.c

\* @author : wangyubin

\* @date : Tue Dec 24 09:49:53 2013

\*

\* @brief : 测试 kobject的创建和删除

\* history : init

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include<linux/init.h>

#include<linux/module.h>

#include<linux/kernel.h>

#include<linux/kobject.h>

MODULE\_LICENSE("Dual BSD/GPL");

struct kobject\* kobj = NULL;

static int test\_kobject\_init(void)

{

/\* 初始化kobject，并加入到sysfs中 \*/

kobj = kobject\_create\_and\_add("test\_kobject", NULL);

/\* 进入内核模块 \*/

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

printk(KERN\_ALERT "test\_kobject is inited!\n");

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

return 0;

}

static void test\_kobject\_exit(void)

{

/\* 如果 kobj 不为空，则将其从sysfs中删除 \*/

if (kobj != NULL)

kobject\_del(kobj);

/\* 退出内核模块 \*/

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

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

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

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

}

module\_init(test\_kobject\_init);

module\_exit(test\_kobject\_exit);

[复制代码](javascript:void(0);)

对应的Makefile

[复制代码](javascript:void(0);)

# must complile on customize kernel

obj-m += mykobject.o

mykobject-objs := test\_kobject.o

#generate the path

CURRENT\_PATH:=$(shell pwd)

#the current kernel version number

LINUX\_KERNEL:=$(shell uname -r)

#the absolute path

LINUX\_KERNEL\_PATH:=/usr/src/kernels/$(LINUX\_KERNEL)

#complie object

all:

make -C $(LINUX\_KERNEL\_PATH) M=$(CURRENT\_PATH) modules

rm -rf modules.order Module.symvers .\*.cmd \*.o \*.mod.c .tmp\_versions \*.unsigned

#clean

clean:

rm -rf modules.order Module.symvers .\*.cmd \*.o \*.mod.c \*.ko .tmp\_versions \*.unsigned

[复制代码](javascript:void(0);)

测试方法：(我使用的测试系统是：Centos6.5 x86)

[复制代码](javascript:void(0);)

[root@localhost test\_kobject]# ll <-- 开始时只有2个文件，一个测试代码，一个Makefile

total 8

-rw-r--r-- 1 root root 533 Dec 24 09:44 Makefile

-rw-r--r-- 1 root root 908 Dec 24 09:44 test\_kobject.c

[root@localhost test\_kobject]# make <-- 编译用于测试的内核模块

make -C /usr/src/kernels/2.6.32-431.el6.i686 M=/home/wyb/chap17/test\_kobject modules

make[1]: Entering directory `/usr/src/kernels/2.6.32-431.el6.i686'

CC [M] /home/wyb/chap17/test\_kobject/test\_kobject.o

LD [M] /home/wyb/chap17/test\_kobject/mykobject.o

Building modules, stage 2.

MODPOST 1 modules

CC /home/wyb/chap17/test\_kobject/mykobject.mod.o

LD [M] /home/wyb/chap17/test\_kobject/mykobject.ko.unsigned

NO SIGN [M] /home/wyb/chap17/test\_kobject/mykobject.ko

make[1]: Leaving directory `/usr/src/kernels/2.6.32-431.el6.i686'

rm -rf modules.order Module.symvers .\*.cmd \*.o \*.mod.c .tmp\_versions \*.unsigned

[root@localhost test\_kobject]# ll <-- 编译后多出来的一个内核模块 \*\*\*.ko

total 100

-rw-r--r-- 1 root root 533 Dec 24 09:44 Makefile

-rw-r--r-- 1 root root 91902 Dec 24 09:54 mykobject.ko

-rw-r--r-- 1 root root 908 Dec 24 09:44 test\_kobject.c

[root@localhost test\_kobject]# ll /sys/ <-- 安装内核模块 mykobject.ko 之前的 sysfs结构

total 0

drwxr-xr-x 2 root root 0 Dec 24 09:28 block

drwxr-xr-x 17 root root 0 Dec 24 09:28 bus

drwxr-xr-x 40 root root 0 Dec 24 09:28 class

drwxr-xr-x 4 root root 0 Dec 24 09:28 dev

drwxr-xr-x 12 root root 0 Dec 24 09:28 devices

drwxr-xr-x 4 root root 0 Dec 24 09:28 firmware

drwxr-xr-x 3 root root 0 Dec 24 09:28 fs

drwxr-xr-x 2 root root 0 Dec 24 09:44 hypervisor

drwxr-xr-x 5 root root 0 Dec 24 09:28 kernel

drwxr-xr-x 84 root root 0 Dec 24 09:46 module

drwxr-xr-x 2 root root 0 Dec 24 09:44 power

[root@localhost test\_kobject]# insmod mykobject.ko <-- 安装内核模块

[root@localhost test\_kobject]# ll /sys/ <-- 安装后，sysfs中多了一个文件夹 test\_kobject

total 0

drwxr-xr-x 2 root root 0 Dec 24 09:28 block

drwxr-xr-x 17 root root 0 Dec 24 09:28 bus

drwxr-xr-x 40 root root 0 Dec 24 09:28 class

drwxr-xr-x 4 root root 0 Dec 24 09:28 dev

drwxr-xr-x 12 root root 0 Dec 24 09:28 devices

drwxr-xr-x 4 root root 0 Dec 24 09:28 firmware

drwxr-xr-x 3 root root 0 Dec 24 09:28 fs

drwxr-xr-x 2 root root 0 Dec 24 09:44 hypervisor

drwxr-xr-x 5 root root 0 Dec 24 09:28 kernel

drwxr-xr-x 85 root root 0 Dec 24 09:54 module

drwxr-xr-x 2 root root 0 Dec 24 09:44 power

drwxr-xr-x 2 root root 0 Dec 24 09:55 test\_kobject

[root@localhost test\_kobject]# ll /sys/test\_kobject/ <-- 追加kobject只能增加文件夹，文件夹中是没有文件的

total 0

[root@localhost test\_kobject]# rmmod mykobject.ko <-- 卸载内核模块

[root@localhost test\_kobject]# ll /sys/ <-- 卸载后，sysfs 中的文件夹 test\_kobject 也消失了

total 0

drwxr-xr-x 2 root root 0 Dec 24 09:28 block

drwxr-xr-x 17 root root 0 Dec 24 09:28 bus

drwxr-xr-x 40 root root 0 Dec 24 09:28 class

drwxr-xr-x 4 root root 0 Dec 24 09:28 dev

drwxr-xr-x 12 root root 0 Dec 24 09:28 devices

drwxr-xr-x 4 root root 0 Dec 24 09:28 firmware

drwxr-xr-x 3 root root 0 Dec 24 09:28 fs

drwxr-xr-x 2 root root 0 Dec 24 09:44 hypervisor

drwxr-xr-x 5 root root 0 Dec 24 09:28 kernel

drwxr-xr-x 84 root root 0 Dec 24 09:55 module

drwxr-xr-x 2 root root 0 Dec 24 09:44 power

[复制代码](javascript:void(0);)

4.1.2. sysfs中添加文件

kobject是映射成sysfs中的目录，那sysfs中的文件是什么呢？

其实sysfs中的文件就是kobject的属性，属性的来源有2个：

+ 默认属性 :: kobject所关联的ktype中的 default\_attrs 字段

默认属性 default\_attrs 的类型是结构体 struct attribute， 定义在 <linux/sysfs.h>

struct attribute {

const char \*name; /\* sysfs文件树中的文件名 \*/

struct module \*owner; /\* x86体系结构中已经不再继续使用了，可能在其他体系结构中还会使用 \*/

mode\_t mode; /\* sysfs中该文件的权限 \*/

};

ktype中的 default\_attrs 字段(即默认属性)描述了sysfs中的文件，还有一个字段 sysfs\_ops 则描述了如何使用默认属性。

struct sysfs\_ops 的定义也在 <linux/sysfs.h>

struct sysfs\_ops {

/\* 在读sysfs文件时该方法被调用 \*/

ssize\_t (\*show)(struct kobject \*kobj, struct attribute \*attr,char \*buffer);

/\* 在写sysfs文件时该方法被调用 \*/

ssize\_t (\*store)(struct kobject \*kobj,struct attribute \*attr,const char \*buffer, size\_t size);

};

show 方法在读取sysfs中文件时调用，它会拷贝attr提供的属性到buffer指定的缓冲区

store  方法在写sysfs中文件时调用，它会从buffer中读取size字节的数据到attr提供的属性中

增加默认属性的示例代码

[复制代码](javascript:void(0);)

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* @file : test\_kobject\_default\_attr.c

\* @author : wangyubin

\* @date : Tue Dec 24 10:28:09 2013

\*

\* @brief : 测试 kobject 的默认属性的创建和删除

\* history : init

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include<linux/init.h>

#include<linux/module.h>

#include<linux/kernel.h>

#include<linux/kobject.h>

#include<linux/sysfs.h>

MODULE\_LICENSE("Dual BSD/GPL");

static void myobj\_release(struct kobject\*);

static ssize\_t my\_show(struct kobject \*, struct attribute \*, char \*);

static ssize\_t my\_store(struct kobject \*, struct attribute \*, const char \*, size\_t);

/\* 自定义的结构体，2个属性，并且嵌入了kobject \*/

struct my\_kobj

{

int ival;

char\* cname;

struct kobject kobj;

};

static struct my\_kobj \*myobj = NULL;

/\* my\_kobj 的属性 ival 所对应的sysfs中的文件，文件名 val \*/

static struct attribute val\_attr = {

.name = "val",

.owner = NULL,

.mode = 0666,

};

/\* my\_kobj 的属性 cname 所对应的sysfs中的文件，文件名 name \*/

static struct attribute name\_attr = {

.name = "name",

.owner = NULL,

.mode = 0666,

};

static int test\_kobject\_default\_attr\_init(void)

{

struct attribute \*myattrs[] = {NULL, NULL, NULL};

struct sysfs\_ops \*myops = NULL;

struct kobj\_type \*mytype = NULL;

/\* 初始化 myobj \*/

myobj = kmalloc(sizeof(struct my\_kobj), GFP\_KERNEL);

if (myobj == NULL)

return -ENOMEM;

/\* 配置文件 val 的默认值 \*/

myobj->ival = 100;

myobj->cname = "test";

/\* 初始化 ktype \*/

mytype = kmalloc(sizeof(struct kobj\_type), GFP\_KERNEL);

if (mytype == NULL)

return -ENOMEM;

/\* 增加2个默认属性文件 \*/

myattrs[0] = &val\_attr;

myattrs[1] = &name\_attr;

/\* 初始化ktype的默认属性和析构函数 \*/

mytype->release = myobj\_release;

mytype->default\_attrs = myattrs;

/\* 初始化ktype中的 sysfs \*/

myops = kmalloc(sizeof(struct sysfs\_ops), GFP\_KERNEL);

if (myops == NULL)

return -ENOMEM;

myops->show = my\_show;

myops->store = my\_store;

mytype->sysfs\_ops = myops;

/\* 初始化kobject，并加入到sysfs中 \*/

memset(&myobj->kobj, 0, sizeof(struct kobject)); /\* 这一步非常重要，没有这一步init kobject会失败 \*/

if (kobject\_init\_and\_add(&myobj->kobj, mytype, NULL, "test\_kobj\_default\_attr"))

kobject\_put(&myobj->kobj);

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

printk(KERN\_ALERT "test\_kobject\_default\_attr is inited!\n");

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

return 0;

}

static void test\_kobject\_default\_attr\_exit(void)

{

kobject\_del(&myobj->kobj);

kfree(myobj);

/\* 退出内核模块 \*/

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

printk(KERN\_ALERT "test\_kobject\_default\_attr is exited!\n");

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

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

}

static void myobj\_release(struct kobject \*kobj)

{

printk(KERN\_ALERT, "release kobject");

kobject\_del(kobj);

}

/\* 读取属性文件 val 或者name时会执行此函数 \*/

static ssize\_t my\_show(struct kobject \*kboj, struct attribute \*attr, char \*buf)

{

printk(KERN\_ALERT "SHOW -- attr-name: [%s]\n", attr->name);

if (strcmp(attr->name, "val") == 0)

return sprintf(buf, "%d\n", myobj->ival);

else

return sprintf(buf, "%s\n", myobj->cname);

}

/\* 写入属性文件 val 或者name时会执行此函数 \*/

static ssize\_t my\_store(struct kobject \*kobj, struct attribute \*attr, const char \*buf, size\_t len)

{

printk(KERN\_ALERT "STORE -- attr-name: [%s]\n", attr->name);

if (strcmp(attr->name, "val") == 0)

sscanf(buf, "%d\n", &myobj->ival);

else

sscanf(buf, "%s\n", myobj->cname);

return len;

}

module\_init(test\_kobject\_default\_attr\_init);

module\_exit(test\_kobject\_default\_attr\_exit);

[复制代码](javascript:void(0);)

对应的Makefile如下：

[复制代码](javascript:void(0);)

# must complile on customize kernel

obj-m += mykobject\_with\_default\_attr.o

mykobject\_with\_default\_attr-objs := test\_kobject\_default\_attr.o

#generate the path

CURRENT\_PATH:=$(shell pwd)

#the current kernel version number

LINUX\_KERNEL:=$(shell uname -r)

#the absolute path

LINUX\_KERNEL\_PATH:=/usr/src/kernels/$(LINUX\_KERNEL)

#complie object

all:

make -C $(LINUX\_KERNEL\_PATH) M=$(CURRENT\_PATH) modules

rm -rf modules.order Module.symvers .\*.cmd \*.o \*.mod.c .tmp\_versions \*.unsigned

#clean

clean:

rm -rf modules.order Module.symvers .\*.cmd \*.o \*.mod.c \*.ko .tmp\_versions \*.unsigned

[复制代码](javascript:void(0);)

测试方法：(我使用的测试系统是：Centos6.5 x86)

[复制代码](javascript:void(0);)

############################ 编译 ########################################################

[root@localhost test\_kobject\_defalt\_attr]# ll

total 8

-rw-r--r-- 1 root root 582 Dec 24 15:02 Makefile

-rw-r--r-- 1 root root 4032 Dec 24 16:58 test\_kobject\_default\_attr.c

[root@localhost test\_kobject\_defalt\_attr]# make

make -C /usr/src/kernels/2.6.32-431.el6.i686 M=/home/wyb/chap17/test\_kobject\_defalt\_attr modules

make[1]: Entering directory `/usr/src/kernels/2.6.32-431.el6.i686'

CC [M] /home/wyb/chap17/test\_kobject\_defalt\_attr/test\_kobject\_default\_attr.o

/home/wyb/chap17/test\_kobject\_defalt\_attr/test\_kobject\_default\_attr.c: In function ‘myobj\_release’:

/home/wyb/chap17/test\_kobject\_defalt\_attr/test\_kobject\_default\_attr.c:109: warning: too many arguments for format

LD [M] /home/wyb/chap17/test\_kobject\_defalt\_attr/mykobject\_with\_default\_attr.o

Building modules, stage 2.

MODPOST 1 modules

CC /home/wyb/chap17/test\_kobject\_defalt\_attr/mykobject\_with\_default\_attr.mod.o

LD [M] /home/wyb/chap17/test\_kobject\_defalt\_attr/mykobject\_with\_default\_attr.ko.unsigned

NO SIGN [M] /home/wyb/chap17/test\_kobject\_defalt\_attr/mykobject\_with\_default\_attr.ko

make[1]: Leaving directory `/usr/src/kernels/2.6.32-431.el6.i686'

rm -rf modules.order Module.symvers .\*.cmd \*.o \*.mod.c .tmp\_versions \*.unsigned

[root@localhost test\_kobject\_defalt\_attr]# ll

total 104

-rw-r--r-- 1 root root 582 Dec 24 15:02 Makefile

-rw-r--r-- 1 root root 96805 Dec 24 16:58 mykobject\_with\_default\_attr.ko

-rw-r--r-- 1 root root 4032 Dec 24 16:58 test\_kobject\_default\_attr.c

############################ 安装 ########################################################

[root@localhost test\_kobject\_defalt\_attr]# insmod mykobject\_with\_default\_attr.ko

[root@localhost test\_kobject\_defalt\_attr]# ll /sys/ <-- kobject对应的文件夹

total 0

drwxr-xr-x 2 root root 0 Dec 24 15:50 block

drwxr-xr-x 17 root root 0 Dec 24 15:50 bus

drwxr-xr-x 40 root root 0 Dec 24 15:50 class

drwxr-xr-x 4 root root 0 Dec 24 15:50 dev

drwxr-xr-x 12 root root 0 Dec 24 15:50 devices

drwxr-xr-x 4 root root 0 Dec 24 15:50 firmware

drwxr-xr-x 3 root root 0 Dec 24 15:50 fs

drwxr-xr-x 2 root root 0 Dec 24 16:06 hypervisor

drwxr-xr-x 5 root root 0 Dec 24 15:50 kernel

drwxr-xr-x 85 root root 0 Dec 24 16:59 module

drwxr-xr-x 2 root root 0 Dec 24 16:06 power

drwxr-xr-x 2 root root 0 Dec 24 16:59 test\_kobj\_default\_attr

[root@localhost test\_kobject\_defalt\_attr]# ll /sys/test\_kobj\_default\_attr/ <-- kobject的2个属性文件

total 0

-rw-rw-rw- 1 root root 4096 Dec 24 16:59 name

-rw-rw-rw- 1 root root 4096 Dec 24 16:59 val

[root@localhost test\_kobject\_defalt\_attr]# dmesg <-- dmesg 中只有初始化的信息

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

test\_kobject\_default\_attr is inited!

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

############################ 读取属性文件 ###############################################

[root@localhost test\_kobject\_defalt\_attr]# cat /sys/test\_kobj\_default\_attr/val <-- 属性值就是我们在测试代码中输入的值

100

[root@localhost test\_kobject\_defalt\_attr]# cat /sys/test\_kobj\_default\_attr/name <-- 属性值就是我们在测试代码中输入的值

test

[root@localhost test\_kobject\_defalt\_attr]# dmesg <-- dmesg 中多了2条读取属性文件的log

SHOW -- attr-name: [val]

SHOW -- attr-name: [name]

############################ 写入属性文件 ################################################

[root@localhost test\_kobject\_defalt\_attr]# echo "200" > /sys/test\_kobj\_default\_attr/val <-- val文件中写入 200

[root@localhost test\_kobject\_defalt\_attr]# echo "abcdefg" > /sys/test\_kobj\_default\_attr/name <-- name文件中写入 adcdefg

[root@localhost test\_kobject\_defalt\_attr]# dmesg <-- dmesg 中又多了2条写入属性文件的log

STORE -- attr-name: [val]

STORE -- attr-name: [name]

[root@localhost test\_kobject\_defalt\_attr]# cat /sys/test\_kobj\_default\_attr/val <-- 再次查看 val文件中的值，已变为200

200

[root@localhost test\_kobject\_defalt\_attr]# cat /sys/test\_kobj\_default\_attr/name <-- 再次查看 name文件中的值，已变为abcdefg

abcdefg

############################ 卸载 ########################################################

[root@localhost test\_kobject\_defalt\_attr]# rmmod mykobject\_with\_default\_attr.ko

[复制代码](javascript:void(0);)

**注**：参考博客 [Linux设备模型 (2)](http://www.cnblogs.com/wwang/archive/2010/12/16/1902721.html)

+ 新属性 :: kobject 自己定义的属性

一般来说，使用默认属性就足够了。因为具有共同ktype的kobject在本质上区别都不大，比如都是块设备的kobject，

它们使用默认的属性集合不但可以让事情简单，有助于代码合并，还可以使类似的对象在sysfs中的外观一致。

在一些特殊的情况下，kobject可能会需要自己特有的属性。内核也充分考虑到了这些情况，提供了创建或者删除新属性的方法。

在sysfs中文件的操作方法参见： fs/sysfs/file.c

[复制代码](javascript:void(0);)

/\* 文件的相关操作非常多，这里只列出创建文件和删除文件的方法 \*/

/\*\*

\* 给 kobj 增加一个新的属性 attr

\* kobj 对应sysfs中的一个文件夹， attr 对应sysfs中的一个文件

\*/

int sysfs\_create\_file(struct kobject \* kobj, const struct attribute \* attr)

{

BUG\_ON(!kobj || !kobj->sd || !attr);

return sysfs\_add\_file(kobj->sd, attr, SYSFS\_KOBJ\_ATTR);

}

/\*\*

\* 给 kobj 删除一个新的属性 attr

\* kobj 对应sysfs中的一个文件夹， attr 对应sysfs中的一个文件

\*/

void sysfs\_remove\_file(struct kobject \* kobj, const struct attribute \* attr)

{

sysfs\_hash\_and\_remove(kobj->sd, attr->name);

}

[复制代码](javascript:void(0);)

除了可以在sysfs中增加/删除的文件，还可以在sysfs中增加或者删除一个符号链接。

具体实现参见：fs/sysfs/symlink.c

[复制代码](javascript:void(0);)

/\* 下面只列出了创建和删除符号链接的方法，其他方法请参考 symlink.c 文件 \*/

/\*\*

\* 在kobj对应的文件夹中创建一个符号链接指向 target

\* 符号链接的名称就是 name

\*/

int sysfs\_create\_link(struct kobject \*kobj, struct kobject \*target,

const char \*name)

{

return sysfs\_do\_create\_link(kobj, target, name, 1);

}

void sysfs\_remove\_link(struct kobject \* kobj, const char \* name)

{

struct sysfs\_dirent \*parent\_sd = NULL;

if (!kobj)

parent\_sd = &sysfs\_root;

else

parent\_sd = kobj->sd;

sysfs\_hash\_and\_remove(parent\_sd, name);

}

[复制代码](javascript:void(0);)

增加新的属性的示例代码 (这里只演示了增加文件的方法，增加符号链接的方法与之类似)

[复制代码](javascript:void(0);)

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* @file : test\_kobject\_new\_attr.c

\* @author : wangyubin

\* @date : Tue Dec 24 17:10:31 2013

\*

\* @brief : 测试 kobject 中增加和删除新属性

\* history : init

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include<linux/init.h>

#include<linux/module.h>

#include<linux/kernel.h>

#include<linux/kobject.h>

#include<linux/sysfs.h>

MODULE\_LICENSE("Dual BSD/GPL");

static void myobj\_release(struct kobject\*);

static ssize\_t my\_show(struct kobject \*, struct attribute \*, char \*);

static ssize\_t my\_store(struct kobject \*, struct attribute \*, const char \*, size\_t);

/\* 自定义的结构体，其中嵌入了kobject，通过属性 c\_attr 来控制增加或者删除新属性 \*/

struct my\_kobj

{

int c\_attr; /\* 值为0：删除新属性， 值为1：增加新属性\*/

int new\_attr;

struct kobject kobj;

};

static struct my\_kobj \*myobj = NULL;

/\* my\_kobj 的属性 c\_attr 所对应的sysfs中的文件，文件名 c\_attr \*/

static struct attribute c\_attr = {

.name = "c\_attr",

.owner = NULL,

.mode = 0666,

};

/\* 用于动态增加或者删除的新属性 \*/

static struct attribute new\_attr = {

.name = "new\_attr",

.owner = NULL,

.mode = 0666,

};

static int test\_kobject\_new\_attr\_init(void)

{

struct attribute \*myattrs[] = {NULL, NULL};

struct sysfs\_ops \*myops = NULL;

struct kobj\_type \*mytype = NULL;

/\* 初始化 myobj \*/

myobj = kmalloc(sizeof(struct my\_kobj), GFP\_KERNEL);

if (myobj == NULL)

return -ENOMEM;

/\* 配置文件 val 的默认值 \*/

myobj->c\_attr = 0;

/\* 初始化 ktype \*/

mytype = kmalloc(sizeof(struct kobj\_type), GFP\_KERNEL);

if (mytype == NULL)

return -ENOMEM;

/\* 增加1个默认属性文件 \*/

myattrs[0] = &c\_attr;

/\* 初始化ktype的默认属性和析构函数 \*/

mytype->release = myobj\_release;

mytype->default\_attrs = myattrs;

/\* 初始化ktype中的 sysfs \*/

myops = kmalloc(sizeof(struct sysfs\_ops), GFP\_KERNEL);

if (myops == NULL)

return -ENOMEM;

myops->show = my\_show;

myops->store = my\_store;

mytype->sysfs\_ops = myops;

/\* 初始化kobject，并加入到sysfs中 \*/

memset(&myobj->kobj, 0, sizeof(struct kobject)); /\* 这一步非常重要，没有这一步init kobject会失败 \*/

if (kobject\_init\_and\_add(&myobj->kobj, mytype, NULL, "test\_kobj\_new\_attr"))

kobject\_put(&myobj->kobj);

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

printk(KERN\_ALERT "test\_kobject\_new\_attr is inited!\n");

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

return 0;

}

static void test\_kobject\_new\_attr\_exit(void)

{

kobject\_del(&myobj->kobj);

kfree(myobj);

/\* 退出内核模块 \*/

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

printk(KERN\_ALERT "test\_kobject\_new\_attr is exited!\n");

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

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

}

static void myobj\_release(struct kobject \*kobj)

{

printk(KERN\_ALERT "release kobject");

kobject\_del(kobj);

}

/\* 读取属性文件 c\_attr 或者 new\_attr 时会执行此函数 \*/

static ssize\_t my\_show(struct kobject \*kboj, struct attribute \*attr, char \*buf)

{

printk(KERN\_ALERT "SHOW -- attr-name: [%s]\n", attr->name);

if (strcmp(attr->name, "c\_attr") == 0)

return sprintf(buf, "%d\n", myobj->c\_attr);

else if (strcmp(attr->name, "new\_attr") == 0)

return sprintf(buf, "%d\n", myobj->new\_attr);

return 0;

}

/\* 写入属性文件c\_attr 或者 new\_attr 时会执行此函数 \*/

static ssize\_t my\_store(struct kobject \*kobj, struct attribute \*attr, const char \*buf, size\_t len)

{

printk(KERN\_ALERT "STORE -- attr-name: [%s]\n", attr->name);

if (strcmp(attr->name, "c\_attr") == 0)

sscanf(buf, "%d\n", &myobj->c\_attr);

else if (strcmp(attr->name, "new\_attr") == 0)

sscanf(buf, "%d\n", &myobj->new\_attr);

if (myobj->c\_attr == 1) /\* 创建新的属性文件 \*/

{

if (sysfs\_create\_file(kobj, &new\_attr))

return -1;

else

myobj->new\_attr = 100; /\* 新属性文件的值默认设置为 100 \*/

}

if (myobj->c\_attr == 0) /\* 删除新的属性文件 \*/

sysfs\_remove\_file(kobj, &new\_attr);

return len;

}

module\_init(test\_kobject\_new\_attr\_init);

module\_exit(test\_kobject\_new\_attr\_exit);

[复制代码](javascript:void(0);)

对应的Makefile如下：

[复制代码](javascript:void(0);)

# must complile on customize kernel

obj-m += mykobject\_with\_new\_attr.o

mykobject\_with\_new\_attr-objs := test\_kobject\_new\_attr.o

#generate the path

CURRENT\_PATH:=$(shell pwd)

#the current kernel version number

LINUX\_KERNEL:=$(shell uname -r)

#the absolute path

LINUX\_KERNEL\_PATH:=/usr/src/kernels/$(LINUX\_KERNEL)

#complie object

all:

make -C $(LINUX\_KERNEL\_PATH) M=$(CURRENT\_PATH) modules

rm -rf modules.order Module.symvers .\*.cmd \*.o \*.mod.c .tmp\_versions \*.unsigned

#clean

clean:

rm -rf modules.order Module.symvers .\*.cmd \*.o \*.mod.c \*.ko .tmp\_versions \*.unsigned

[复制代码](javascript:void(0);)

测试方法：(我使用的测试系统是：Centos6.5 x86)

根据测试代码，增加/删除新属性是根据默认属性 c\_attr 的值来决定的。

c\_attr 设置为0 时，删除新属性 new\_attr

c\_attr 设置为1 时，新增新属性 new\_attr

[复制代码](javascript:void(0);)

############################ 编译，安装，卸载同测试默认属性时一样 #######################

... 省略 ...

############################ 动态增加新属性文件 #########################################

[root@localhost test\_kobject\_new\_attr]# ll /sys/test\_kobj\_new\_attr/ <-- 默认没有新属性 new\_attr

total 0

-rw-rw-rw- 1 root root 4096 Dec 24 18:47 c\_attr

[root@localhost test\_kobject\_new\_attr]# cat /sys/test\_kobj\_new\_attr/c\_attr <-- c\_attr 的值为0

0

[root@localhost test\_kobject\_new\_attr]# echo "1" > /sys/test\_kobj\_new\_attr/c\_attr <-- c\_attr 的值设为1

[root@localhost test\_kobject\_new\_attr]# ll /sys/test\_kobj\_new\_attr/ <-- 增加了新属性 new\_attr

total 0

-rw-rw-rw- 1 root root 4096 Dec 24 19:02 c\_attr

-rw-rw-rw- 1 root root 4096 Dec 24 19:02 new\_attr

############################ 动态删除属性文件 ###########################################

[root@localhost test\_kobject\_new\_attr]# echo "0" > /sys/test\_kobj\_new\_attr/c\_attr <-- c\_attr 的值为0

[root@localhost test\_kobject\_new\_attr]# ll /sys/test\_kobj\_new\_attr/ <-- 删除了新属性 new\_attr

total 0

-rw-rw-rw- 1 root root 4096 Dec 24 19:03 c\_attr

[复制代码](javascript:void(0);)

4.1.3. sysfs相关约定

为了保持sysfs的干净和直观，在内核开发中涉及到sysfs相关内容时，需要注意以下几点：

+ sysfs属性保证每个文件只导出一个值，该值为文本形式并且可以映射为简单的C类型

+ sysfs中要以一个清晰的层次组织数据

+ sysfs提供内核到用户空间的服务

#### 4.2 基于sysfs的内核事件

内核事件层也是利用kobject和sysfs来实现的，用户空间通过监控sysfs中kobject的属性的变化来异步的捕获内核中kobject发出的信号。

用户空间可以通过一种netlink的机制来获取内核事件。

内核空间向用户空间发送信号使用 kobject\_uevent() 函数，具体参见： <linux/kobject.h>

int kobject\_uevent(struct kobject \*kobj, enum kobject\_action action);

下面用个小例子演示一些内核事件的实现原理：

4.2.1. 内核模块安装或者删除时，会发送 KOBJ\_ADD 或者 KOBJ\_REMOVE 的消息

内核模块的代码就用上面最简单的那个例子 test\_kobject.c 的代码即可

4.2.2. 用户态程序: 通过 netlink机制来接收 kobject 的事件通知

[复制代码](javascript:void(0);)

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* @file : test\_netlink\_client.c

\* @author : wangyubin

\* @date : Tue Dec 24 19:48:54 2013

\*

\* @brief : 通过 netlink机制接收kobject发出的信号

\* history : init

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <errno.h>

#include <sys/types.h>

#include <asm/types.h>

#include <sys/socket.h>

#include <linux/netlink.h>

void MonitorNetlinkUevent()

{

int sockfd;

struct sockaddr\_nl sa;

int len;

char buf[4096];

struct iovec iov;

struct msghdr msg;

int i;

memset(&sa,0,sizeof(sa));

sa.nl\_family = AF\_NETLINK;

sa.nl\_groups = NETLINK\_KOBJECT\_UEVENT;

sa.nl\_pid = 0;//getpid(); both is ok

memset(&msg,0,sizeof(msg));

iov.iov\_base = (void \*)buf;

iov.iov\_len = sizeof(buf);

msg.msg\_name = (void \*)&sa;

msg.msg\_namelen = sizeof(sa);

msg.msg\_iov = &iov;

msg.msg\_iovlen = 1;

sockfd = socket(AF\_NETLINK, SOCK\_RAW, NETLINK\_KOBJECT\_UEVENT);

if(sockfd == -1)

printf("socket creating failed:%s\n",strerror(errno));

if(bind(sockfd,(struct sockaddr \*)&sa,sizeof(sa)) == -1)

printf("bind error:%s\n", strerror(errno));

while(1) {

memset(buf, 0, sizeof(buf));

len=recvmsg(sockfd, &msg, 0);

if(len < 0){}

//printf("receive error\n");

else if(len < 32||len > sizeof(buf))

printf("invalid message");

for(i=0; i<len; i++)

if(\*(buf+i) == '\0')

buf[i] = '\n';

printf("received %d bytes\n%s\n", len, buf);

}

}

int main(int argc, char \*argv[])

{

MonitorNetlinkUevent();

return 0;

}

[复制代码](javascript:void(0);)

**注**：代码是拷贝的 [《Linux设备节点创建》内核kobject上报uevent过滤规则](http://blog.csdn.net/tankai19880619/article/details/11776589) 中的用户态程序部分

4.2.3. 测试方法：(我使用的测试系统是：Centos6.5 x86)

[复制代码](javascript:void(0);)

############################ 编译并启动用户态程序 （窗口1）###################################

[root@localhost test\_kobject\_event]# ll

total 4

-rw-r--r-- 1 root root 1846 Dec 24 20:36 test\_netlink\_client.c

[root@localhost test\_kobject\_event]# gcc -o test\_netlink\_client test\_netlink\_client.c <-- 编译用户态程序

[root@localhost test\_kobject\_event]# ./test\_netlink\_client <-- 启动后等待内核kobject的事件到来

############################ 安装内核模块并查看用户态程序输出 （窗口2）#######################

[root@localhost test\_kobject]# insmod mykobject.ko <-- 在窗口2中安装内核模块，窗口1中会接收到 KOBJ\_ADD 信号

############################ 卸载内核模块并查看用户态程序输出 （窗口2）#######################

[root@localhost test\_kobject]# rmmod mykobject.ko <-- 在窗口2中安装内核模块，窗口1中会接收到 KOBJ\_REMOVE 信号

[复制代码](javascript:void(0);)

### ****5. 总结****

kobject加sysfs给内核带来的好处不是三言两语能够说得清楚的，上面的例子也只是通过简单的使用来直观的感受一下kobject，例子本身没有什么实际意义。

最后一个例子中使用了 netlink机制，我在之前的项目中使用过netlink来监控系统进程的I/O信息，对netlink有一些初步的了解。

但是例子中只是用来获取一下 kobject的事件而已，这里主要为了说明kobject，netlink的相关内容以后有机会再补充

# [内核调试](http://www.cnblogs.com/wang_yb/p/3504539.html)

内核调试的难点在于它不能像用户态程序调试那样打断点，随时暂停查看各个变量的状态。

也不能像用户态程序那样崩溃后迅速的重启，恢复初始状态。

用户态程序和内核交互，用户态程序的各种状态，错误等可以由内核来捕获并显示。

而内核是直接和硬件交互的，内核出错之后整个系统就无法正常运行了，所以要想熟练的进行内核调试，

首先要熟悉内核已经给我们提供的工具，然后实实在在的去做一些内核功能的开发，在开发的过程中不断熟悉内核代码，增加内核调试的经验。

**主要内容：**

* 内核调试的难点
* 内核调试的工具和方法
* 总结

### ****1. 内核调试的难点****

内核调试的难点大致有以下几个：

1. 重现bug困难 - 如果能够重现一个bug, 相当于成功了一半. (特别是有些bug和硬件相关, 执行几百万次之后才有一次错误)
2. 调试风险比较大 - 稍有不慎, 即造成系统崩溃
3. 定位bug的初始版本困难 - 内核版本更新很快, 很难确定bug是在那个版本开始出现的

### ****2. 内核调试的工具和方法****

内核调试虽然困难, 但同时也极具挑战性, 如果能够解决一个困扰大家多时的内核bug, 那将会给自己带来极大的成就感. :)

而且, 随着内核的不断发展, 内核调试的手段和方法也在不断进步, 下面是书中提到的一些常用的调试手段.

#### 2.1 输出 LOG

输出LOG不光是内核调试, 即使是在用户态程序的调试中, 也是经常使用的一个调试手段.

通过在可疑的代码周围加上一些LOG输出, 可以准确的了解bug发生前后的一些重要信息.

##### 2.1.1 LOG等级

linux内核中输出LOG的函数是 printk (语法和printf几乎雷同, 唯一的区别是printk可以指定日志级别)

printk之所以好用, 就在与它随时都可以被调用, 没有任何限制条件.

printk的输出日志级别如下:

|  |  |
| --- | --- |
| **等级** | **描述** |
| KERN\_EMERG | 一个紧急情况 |
| KERN\_ALERT | 一个需要立即被注意到的错误 |
| KERN\_CRIT | 一个临界情况 |
| KERN\_ERR | 一个错误 |
| KERN\_WARNING | 一个警告 |
| KERN\_NOTICE | 一个普通的, 不过也有可能需要注意的情况 |
| KERN\_INFO | 一条非正式的消息 |
| KERN\_DEBUG | 一条调试信息--一般是冗余信息 |

输出示例:

printk(KERN\_WARNING "This is a warning!\n");

printk(KERN\_DEBUG "This is a debug notice!\n");

##### 2.1.2 LOG记录

标准linux系统上, printk 输出log之后, 由用户空间的守护进程klogd从缓冲区中读取内核消息, 然后再通过syslogd守护进程将它们保存在系统日志文件中.

syslogd 将接受到的所有内核消息添加到一个文件中, 该文件默认为: /var/log/dmesg (系统Centos6.4 x86\_64)

PS. 上篇博客中的内核模块的输出LOG, 都可以在 /var/log/dmesg 中看到

#### 2.2 oops

oopss是个拟声词, 类似 "哎哟" 的意思. 它是内核通知用户有不幸发生的最常用方式.

触发一个oops很简单, 其实只要在上篇博客中的那些内核模块示例中随便找一个, 里面加上一段给未初始化的指针赋值的代码, 就能触发一个oops

oops中包含错误发生时的一些重要信息(比如, 寄存器上下文和回溯线索), 对调试bug很有帮助!

调试内核时, 还可以开启内核编译参数中的各种和内核调试相关的选项, 那样还可以给我们提供内核崩溃时的一些额外信息.

#### 2.3 主动触发bug

调试中有时将某些情况下标记为bug, 执行到这些情况时, 提供断言并输出信息.

BUG 和 BUG\_ON 就是2个可以主动触发oops的内核调用.

在不应该被执行到的地方使用 BUG 或者 BUG\_ON 来捕获.

比如:

if (bad\_thing)

BUG();

// 或者

BUG\_ON(bad\_thing);

如果想要触发更为严重的错误, 可以使用 panic() 函数

比如:

if (terrible\_thing)

panic("terrible thing is %ld\n", terrible\_thing);

此外, 还有dump\_stack 函数可以打印寄存器上下文和回溯信息.

比如:

if (!debug\_check) {

printk(KERN\_DEBUG "provide some information...\n");

dump\_statck();

}

#### 2.4 神奇的系统请求键

这个系统请求键之所以神奇, 在于它可以在一个快挂了的系统上输出一些有用的信息.

这个按键一般就是标准键盘上的 [SysRq] 键 (就在 F12 键右边, 其实就是windows中截整个屏幕的按键)

单独按那个键相当于截屏, 按住 ALT + [SysRq] = [SysRq]的功能

启用这个键的功能有2个方法:

* 开启内核编译选项 : CONFIG\_MAGIC\_SYSRQ
* 动态启用: echo 1 > /proc/sys/kernel/sysrq

支持 SysRq 的命令如下: (注意要在控制台界面下使用这个键, 比如通过 ALT+CTRL+F2 进入一个控制台界面)

|  |  |
| --- | --- |
| **主要命令** | **描述** |
| SysRq-b | 重新启动机器 |
| SysRq-e | 向init以外的所有进程发送SIGTERM信号 |
| SysRq-h | 在控制台显示SysRq的帮助信息 |
| SysRq-i | 向init以外的所有进程发送SIGKILL信号 |
| SysRq-k | 安全访问键:杀死这个控制台上的所有程序 |
| SysRq-l | 向包括init的所有进程发送SIGKILL信号 |
| SysRq-m | 把内存信息输出到控制台 |
| SysRq-o | 关闭机器 |
| SysRq-p | 把寄存器信息输出到控制台 |
| SysRq-r | 关闭键盘原始模式 |
| SysRq-s | 把所有已安装文件系统都刷新到磁盘 |
| SysRq-t | 把任务信息输出到控制台 |
| SysRq-u | 卸载所有已加载文件系统 |

#### 2.5 内核调试器 gdb和kgdb

linux内核的调试器可以使用 gdb或者kgdb, 配置比较麻烦, 准备实际用调试的时候再去试试效果如何..

#### 2.6 探测系统

下面一些方法是在修改内核后, 用来试探内核反应的小技巧.

##### 2.6.1 用UID控制内核执行

比如在内核中加入了新的特性, 为了测试特性, 可以用UID来控制内核是否执行新特性.

if (current->uid != 7777) {

/\* 原先的代码 \*/

} else {

/\* 新的特性 \*/

}

##### 2.6.2 用条件变量控制内核执行

也可以设置一些条件变量来控制内核是否执行某段代码.

条件变量可以像上篇博客中那样, 设置在 sys 文件系统的某个文件中. 当文件中的值变化时, 通知内核执行相应的代码.

##### 2.6.3 使用统计量观察内核执行某段代码的频率

实现思路就是在内核中的设置一个全局变量, 比如 my\_count, 当内核执行到某段代码时, 给 my\_count + 1 就行.

同时还要将 my\_count 打印出来(可以用printk), 便于随时查看它的值.

##### 2.6.4 控制内核执行某段代码的频率

有时侯, 我们需要在内核发生错误时打印错误相关的信息, 如果这个错误不会导致内核崩溃, 并且这个错误每秒会发生几百次甚至更多.

那么, 用printk输出的信息会非常多, 给系统造成额外的负担.

这时, 我们就需要想办法控制错误输出的频率, 有2种方法:

**方法1:** 隔一段时间才输出一次错误

static unsigned long prev\_jiffy = jiffies; /\* 频率限制 \*/

if (time\_after(jiffies, prev\_jiffy + 2\*HZ)) { /\* 输出间隔至少 2HZ \*/

prev\_jiffy = jiffies;

printk(KERN\_ERR "错误信息....\n");

}

**方法2:** 输出 N 次之后, 不再输出(N是正整数)

static unsigned long limit = 0;

if (limit < 5) { /\* 输出5次错误信息后就不再输出 \*/

limit++;

printk(KERN\_ERR "错误信息....\n");

}

#### 2.7 二分法查找bug发生的最初内核版本

在内核发生了bug之后, 如果能够知道是bug从哪个内核版本开始出现的, 那对修正这个bug会有很大的帮助.

由于内核代码非常庞大, 即使用二分查找法, 手工去找哪个版本开始出现bug的话, 仍然是非常耗时和繁琐的.

好在 Git 给我们提供了一个非常有用的二分搜索机制.

git bisect start # 开始二分搜索

git bisect bad <bad\_revision> # 指定一个bug出现的内核版本号

git bisect good <good\_revision> # 指定一个没有bug的内核版本号, 此时git会检测2个版本直接的隐患

# 根据结果再次设置 bad 和 good 的版本号, 缩小Git检索范围, 直至找到可疑之处为止.

#### 2.8 社区

当你在调试bug时用尽了一切手段仍然无济于事时, 可以考虑求助linux社区, 求助时注意一定要描述清楚bug的状况.

(可以参考一下别人汇报bug的格式)

linux 内核相关的邮件列表非常多: 参见 <http://vger.kernel.org/vger-lists.html>

和内核开发, bug汇报相关的邮件列表参见: <http://vger.kernel.org/vger-lists.html#linux-kernel> (这个邮件列表非常活跃, 每天的邮件都非常多!!!)

### ****3. 总结****

linux内核调试必须要依靠大量的实践来掌握, 仅仅靠上面介绍的一些技巧还远远不够, 只有实实在在的去阅读内核代码, 实实在在的去修正一个个bug, 才算真正掌握内核, 真正了解内核.

多看看之前linux内核bug的修正案例, 也是个不错的积累经验的方法.

PS.

对于初学者来说, 在真机上做内核开发动辄导致机器崩溃(panic), 非常麻烦. 现在的虚拟机这么强大, 建议都在虚拟机上测试linux内核修改的效果.

我之前的关于<<Linux内核设计与实现>>笔记的博客中的代码都是在虚拟机上运行测试的.

# [可移植性](http://www.cnblogs.com/wang_yb/p/3512095.html)

linux内核的移植性非常好, 目前的内核也支持非常多的体系结构(有20多个).

但是刚开始时, linux也只支持 intel i386 架构, 从 v1.2版开始支持 Digital Alpha, Intel x86, MIPS和SPARC(虽然支持的还不是很完善).

从 v2.0版本开始加入了对 Motorala 68K和PowerPC的官方支持, v2.2版本开始新增了 ARMS, IBM S390和UltraSPARC的支持.

v2.4版本支持的体系结构数达到了15个, v2.6版本支持的体系结构数目提高到了21个.

目前的我使用的系统是 Fedora20, 支持的体系结构有31个之多.(源码树中 arch目录下有支持的体系结构, 每种体系结构一个文件夹)

考虑到内核支持如此之多的架构, 在内核开发的时候就需要考虑编码的可移植性.

提高可移植性最重要的就是要搞明白不同体系结构之间究竟是什么对移植代码的影响比较大.

**主要内容：**

* 字长
* 数据类型
* 数据对齐
* 字节顺序
* 时间
* 页长度
* 处理器顺序
* SMP, 内核抢占, 高端内存
* 总结

### ****1. 字长****

这里的字是指处理器能够一次完成处理的数据. 字长即使处理器能够一次完成处理的数据的最大长度.

目前的处理器主要有32位和64为2种, 注意这里的32位和64位并不是指操作系统的版本, 而是指处理器的能力.

一般来说, 32位的处理器只能安装32位的操作系统, 而64位的处理器可以安装32位的操作系统, 也可以安装64位的操作系统.

对于一种体系结构来说, 处理器通用寄存器(general-purpose registers, GPR)的大小和它的字长是相同的.

C语言定义的long类型总是对等于机器的字长, 而int型有时会比字长小.

* 32位的体系结构中, int型和long型都是32位的
* 64位的体系结构中, int型是32位的, long型是64位的.

内核编码中涉及到字长的部分时, 牢记以下准则:

1. ANSI C标准规定, 一个char的长度一定是一个字节(8位)
2. linux当前所支持的体系结构中, int型都是32位的
3. linux当前所支持的体系结构中, short型都是16位的
4. linux当前所支持的体系结构中, 指针和long型的长度不定, 在32位和64位中变化
5. 不能假设 sizeof(int) == sizeof(long)
6. 类似的, 不能假定 指针的长度和int型相同.

此外, 操作系统有个简单的助记符来描述此系统中数据类型的大小.

* LLP64 :: 64位的Windows, long类型和指针都是64位
* LP64 :: 64位的Linux, long类型和指针都是64位
* ILP32 :: 32位的Linux, int类型, long类型和指针都是32位
* ILP64 :: int类型, long类型和指针都是64位(非Linux)

### ****2. 数据类型****

编写可移植性代码时, 内核中的数据类型有以下3点需要注意:

#### 2.1 不透明类型

linux内核中定义了很多不透明类型, 它们是在C语言标准类型上的一个封装, 比如 pid\_t, uid\_t, gid\_t 等等.

例如, pid\_t的定义可以在源码中找到:

typedef \_\_kernel\_pid\_t pid\_t; /\* include/linux/types.h \*/

typedef int \_\_kernel\_pid\_t; /\* arch/asm/include/asm/posix\_types.h \*/

使用这些不透明类型时, 以下原则需要注意:

1. 不要假设该类型的长度(那怕通过源码看到了它的C语言类型), 这些类型在不同体系结构中可能长度会变, 内核开发者也有可能修改它们
2. 不要将这些不透明类型转换为C标准类型来使用
3. 编程时保证不透明类型实际存储空间或者格式发生变化时代码不受影响

#### 2.2 长度确定的类型

除了不透明类型, linux内核中还定义了一系列长度明确的数据类型, 参见 include/asm-generic/int-l64.h 或者 include/asm-generic/int-ll64.h

[复制代码](javascript:void(0);)

typedef signed char s8;

typedef unsigned char u8;

typedef signed short s16;

typedef unsigned short u16;

typedef signed int s32;

typedef unsigned int u32;

typedef signed long s64;

typedef unsigned long u64;

[复制代码](javascript:void(0);)

上面这些类型只能在内核空间使用, 用户空间无法使用. 用户空间有对应的变量类型, 名称前多了2个下划线:

[复制代码](javascript:void(0);)

typedef \_\_signed\_\_ char \_\_s8;

typedef unsigned char \_\_u8;

typedef \_\_signed\_\_ short \_\_s16;

typedef unsigned short \_\_u16;

typedef \_\_signed\_\_ int \_\_s32;

typedef unsigned int \_\_u32;

typedef \_\_signed\_\_ long \_\_s64;

typedef unsigned long \_\_u64;

[复制代码](javascript:void(0);)

#### 2.3 char类型

之所以把char类型单独拿出来说明, 是因为char类型在不同的体系结构中, 有时默认是带符号的, 有时是不带符号的.

比如, 最简单的例子:

/\*

\* 某些体系结构中, char类型默认是带符号的, 那么下面 i 的值就为 -1

\* 某些体系结构中, char类型默认是不带符号的, 那么下面 i 的值就为 255, 与预期可能有差别!!!

\*/

char i = -1;

避免上述问题的方法就是, 给char类型赋值时, 明确是否带符号, 如下:

signed char i = -1; /\* 明确 signed, i 的值在哪种体系结构中都是 -1 \*/

unsigned char i = 255; /\* 明确 unsigned, i 的值在哪种体系结构中都是 255 \*/

### ****3. 数据对齐****

数据对齐也是增强可移植性的一个重要方面(有的体系结构对数据对齐要求非常严格, 载入未对齐的数据可导致性能下降, 甚至错误).

数据对齐的意思就是: 数据的内存地址可以被 4 整除

1. 通过指针转换类型时, 不要转换长度不一样的类型, 比如下面的代码有可能出错

[复制代码](javascript:void(0);)

/\*

\* 下面的代码将一个变量从 char 类型转换为 unsigned long 类型,

\* char 类型只占 1个字节, 它的地址不一定能被4整除, 转换为 4个字节或者8个字节的 usigned long之后,

\* 导致 unsigned long 出现数据不对齐的现象.

\*/

char wolf[] = "Like a wolf";

char \*p = &wolf[1];

unsigned long p1 = \*(unsigned long\*) p;

[复制代码](javascript:void(0);)

2. 对于数组, 安装基本数据类型进行对齐就行.(数组元素的存放在内存中是连续的, 第一个对齐了, 后面的都自动对齐了)

3. 对于联合体, 长度最大的数据对齐就可以了

4. 对于结构体, 保证结构体中每个元素能够正确对齐即可

如果结构体中的元素没有对齐, 编译器会自动填充结构体, 保证它是对齐的. 比如下面的代码, 预计应该输出**12**, 实际却输出了**24**

我的代码运行环境: Fedora20 x86\_64

[复制代码](javascript:void(0);)

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* @file : struct\_align.c

\* @author : wangyubin

\* @date : 2014-01-09

\*

\* @brief :

\* history : init

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <stdio.h>

struct animal\_struct

{

char dog; /\* 1个字节 \*/

unsigned long cat; /\* 8个字节 \*/

unsigned short pig; /\* 2个字节 \*/

char fox; /\* 1个字节 \*/

};

int main(int argc, char \*argv[])

{

/\* 在我的64bit 系统中是按8位对齐, 下面的代码输出 24 \*/

printf ("sizeof(animal\_struct)=%d\n", sizeof(struct animal\_struct));

return 0;

}

[复制代码](javascript:void(0);)

测试方法:

gcc -o test struct\_align.c

./test # 输出24

结构体应该被填充成如下形式:

[复制代码](javascript:void(0);)

struct animal\_struct

{

char dog; /\* 1个字节 \*/

/\* 此处填充了7个字节 \*/

unsigned long cat; /\* 8个字节 \*/

unsigned short pig; /\* 2个字节 \*/

char fox; /\* 1个字节 \*/

/\* 此处填充了5个字节 \*/

};

[复制代码](javascript:void(0);)

通过调整结构体中元素顺序, 可以减少填充的字节数, 比如上述结构体如果定义成如下顺序:

[复制代码](javascript:void(0);)

struct animal\_struct

{

unsigned long cat; /\* 8个字节 \*/

unsigned short pig; /\* 2个字节 \*/

char dog; /\* 1个字节 \*/

char fox; /\* 1个字节 \*/

};

[复制代码](javascript:void(0);)

那么为了保证8位对齐, 只需在后面补充 4位即可:

[复制代码](javascript:void(0);)

struct animal\_struct

{

unsigned long cat; /\* 8个字节 \*/

unsigned short pig; /\* 2个字节 \*/

char dog; /\* 1个字节 \*/

char fox; /\* 1个字节 \*/

/\* 此处填充了4个字节 \*/

};

[复制代码](javascript:void(0);)

调整后的代码会输出 16, 不是之前的24

[复制代码](javascript:void(0);)

#include <stdio.h>

struct animal\_struct

{

unsigned long cat; /\* 8个字节 \*/

unsigned short pig; /\* 2个字节 \*/

char dog; /\* 1个字节 \*/

char fox; /\* 1个字节 \*/

};

int main(int argc, char \*argv[])

{

/\* 在我的64bit 系统中是按8位对齐, 下面的代码输出 16 \*/

printf ("sizeof(animal\_struct)=%d\n", sizeof(struct animal\_struct));

return 0;

}

[复制代码](javascript:void(0);)

测试方法:

gcc -o test struct\_align.c

./test # 输出16

**注意:** 虽然调整结构体中元素的顺序可以减少填充的字节, 从而降低内存的消耗.

但是对于内核中已有的那些结构, 千万不能随便调整其元素顺序, 因为内核中很多现存的方法都是通过元素在结构体中位置偏移来获取元素的.

### ****4. 字节顺序****

字节顺序其实只有2种:

* 低位优先 :: little-endian 数据由低位地址->高位地址存放
* 高位优先 :: big-endian 数据由高位地址->低位地址存放

比如占有四个字节的整数的二进制表示如下:

00000001 00000002 00000003 00000004

内存地址方向:   高位  <--------------------> 低位

little-endian 表示如下:

00000001 00000002 00000003 00000004

big-endian 表示如下:

00000004 00000003 00000002 00000001

判断一个体系结构是 big-endian 还是 little-endian 非常简单.

[复制代码](javascript:void(0);)

int x = 1; /\* 二进制 00000000 00000000 00000000 00000001 \*/

/\*

\* 内存地址方向: 高位 <--------------------> 低位

\* little-endian 表示: 00000000 00000000 00000000 00000001

\* big-endian 表示: 00000001 00000000 00000000 00000000

\*/

if (\*(char \*) &x == 1) /\* 这句话把int型转为char型, 相当于只取了int型的最低8bit \*/

/\* little-endian \*/

else

/\* big-endian \*/

[复制代码](javascript:void(0);)

### ****5. 时间****

内核中使用到时间相关概念时, 为了提高可移植性, 不要使用时间中断的发生频率(也就是每秒产生的jiffies), 而应该使用 HZ 来正确使用时间.

关于 jiffies 和 HZ 的概念, 可以参考之前的博客: [《Linux内核设计与实现》读书笔记（十一）- 定时器和时间管理](http://www.cnblogs.com/wang_yb/archive/2013/05/10/3070373.html)

### ****6. 页长度****

当处理用页管理的内存时, 不要既定页的长度为 4KB, 在不同的体系结构中长度会不一样.

而应该使用 **PAGE\_SIZE** 以字节数来表示页长度, 使用 **PAGE\_SHIFT** 表示从最右端屏蔽了多少位能够得到该地址对应的页的页号.

**PAGE\_SIZE** 和 **PAGE\_SHIFT** 都是宏, 定义在 include/asm-generic/page.h 中

下表是一些体系结构中页长度:

|  |  |  |
| --- | --- | --- |
| **体系结构** | **PAGE\_SHIFT** | **PAGE\_SIZE** |
| alpha | 13 | 8KB |
| arm | 12, 14, 15 | 4KB, 16KB, 32KB |
| avr | 12 | 4KB |
| cris | 13 | 8KB |
| blackfin | 12 | 16KB |
| h8300 | 14 | 4KB |
|  | 12 | 4KB, 8KB, 16KB, 32KB |
| m32r | 12, 13, 14, 16 | 4KB |
| m68k | 12 | 4KB, 8KB |
| m68knommu | 12, 13 | 4KB |
| mips | 12 | 4KB |
| min10300 | 12 | 4KB |
| parisc | 12 | 4KB |
| powerpc | 12 | 4KB |
| s390 | 12 | 4KB |
| sh | 12 | 4KB |
| sparc | 12, 13 | 4KB, 8KB |
| um | 12 | 4KB |
| x86 | 12 | 4KB |
| xtensa | 12 | 4KB |

### ****7. 处理器顺序****

还有最后一个和可移植性相关的注意点就是处理器对代码的执行顺序, 在有些体系结构中, 处理器并不是严格按照代码编写的顺序执行的,

可能为了优化性能或者其他原因, 处理器执行指令的顺序与编写的代码的顺序稍有出入.

如果我们的某段代码需要严格的执行顺序, 需要在代码中使用 rmb() wmb() 等内存屏障来确保处理器的执行顺序.

关于rmb和wmb可以参考之前的博客: [《Linux内核设计与实现》读书笔记（十）- 内核同步方法](http://www.cnblogs.com/wang_yb/archive/2013/05/01/3052865.html)  第 11 小节

### ****8. SMP, 内核抢占, 高端内存****

SMP, 内核抢占和高端内存本身虽然和可移植性没有太大的关系, 但它们都是内核中重要的配置选项,

如果编码时能够考虑到这些的话, 那么即使内核修改SMP等这些配置选项, 我们的代码仍然可以安全可靠的运行.

所以, 在编写内核代码时最好加上如下假设:

* 假设代码会在SMP系统上运行, 要正确选择和使用锁
* 假设代码会在支持内核抢占的情况下运行, 要正确使用锁和内核抢占语句
* 假设代码会运行在使用高端内存(非永久映射内存)的系统上, 必要时使用 kmap()

### ****9. 总结****

编写简洁, 可移植性的代码还需要通过实践来积累经验, 上面的准则可以作为代码是否满足可移植性的一些检测条件.

书中还提到的2点注意事项, 我觉得不仅是编写内核代码, 编写任何代码时, 都应该注意:

* 编码尽量选取最大公因子 :: 假定任何事情都有可能发生, 任何潜在的约束也都存在
* 编码尽量选取最小公约数 :: 不要假定给定的内核特性是可用的, 仅仅需要最小的体系结构功能

**虽然编写可移植性代码需要遵守这么多的原则, 但是不能畏惧, 在学习内核开发的过程中, 只有不断的尝试, 不断的犯错, 才能确实的掌握内核**

# [（二十）- 补丁, 开发和社区](http://www.cnblogs.com/wang_yb/p/3514701.html)

linux最吸引我的地方之一就是它拥有一个高手云集的社区, 还有就是如果能=为linux内核中贡献代码, 一定是一件令人自豪的事情.

下面主要总结一些和贡献代码相关的主要内容.

* 加入社区
* 编码风格
* 提交补丁
* 总结

### ****1. 加入社区****

如果想为linux贡献代码, 那么加入linux社区是必须的, 加入了社区, 不仅可以及时内核的最新消息, 而且可以及时和社区内有经验的内核开发者交流经验.

同时也是提交代码和讨论代码的地方, 了解社区的规则, 融入社区环境之中, 才能更好的学习内核, 体会内核开发的乐趣和成就感.

内核社区说白了就是内核邮件列表(LKML linux kernel mail list)

订阅邮件列表的网址: <http://vger.kernel.org/vger-lists.html> 这里面有linux相关的各种邮件列表

关于内核的邮件列表是:  <http://vger.kernel.org/vger-lists.html#linux-kernel>

除了邮件列表之外, 还有2个本书作者推荐的网站也适合linux内核新手去关注:

1. [http://kernelnewbies.org](http://kernelnewbies.org/) 有很多适合内核开发入门的资源
2. [http://lwn.net](http://lwn.net/) linux 新闻周刊

### ****2. 编码风格****

社区给我们提供了学习和贡献内核的地方, 但是为了避免不必要的麻烦(被别人指责或者无人理睬), 首先得好好了解一些内核代码的编码风格.

linux的编码风格都记录在 Documentation/CodingStyle 内核开发前要好好研读以下, 之后有时间我会整理到博客中.

### ****3. 提交补丁****

准备工作都完成之后, 就可以开始内核开发之旅了 :)

只要坚持不断的学习和尝试, 总有一天会为了内核贡献自己的代码, 这时候, 就需要了解如何提交代码, 也就是内核补丁.

如果是发现了BUG或者有改善, 可以将BUG的描述或者改善代码发送给对应的维护者.(内核各个子系统的维护者信息在内核代码根目录下的 MAINTAINERS 文件中)

生成BUG或者改善代码的补丁有2种方法:

**1. 用diff命令创建补丁**

[复制代码](javascript:void(0);)

# 生成patch

diff -urN linux-old/ linux-new/ > my-patch # 比对整个内核代码文件夹

OR

diff -u linux-old/some/file linux-new/some/file > my-patch # 比对某个文件

# 应用patch, 应用了patch之后, linux-old 和 linux-new 中的代码就一样了

cd linux-old

patch -p1 < ../my-patch # 这个命令是进入linux内核代码根目录内执行的

# PS. 还有个很有用的工具 diffstat

diffstat -p1 my-patch # 列出补丁所引起的变更的统计(加入或移去的代码行)

[复制代码](javascript:void(0);)

**2. 用git命令创建补丁**

[复制代码](javascript:void(0);)

# 提交修改的或新增的代码

git commit -a # 提交所有修改的代码

OR

git commit linux-src/some/file.c # 提交某个修改的代码

OR

git add linux-src/some/new-file.c # 把新增的文件加入版本库

git commit -a # 提交新增的文件

# 生成patch

git format-patch -N # N 是正整数, 这条命令生成最后N次提交产生的补丁

OR

git format-patch -1 # 最后1次提交产生的补丁

# 应用patch: 和第一种方法一样

[复制代码](javascript:void(0);)

### ****4. 总结****

本章的内容都是和提交内核patch有关, 我还是内核的入门者, 没有社区的经验, 更别说提交内核patch的经验了.

这篇笔记只是简单记录一些对入门者有用的信息, 便于以后查看.

# 参考

https://blog.csdn.net/weixin\_30586085/article/details/101833391