# [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)

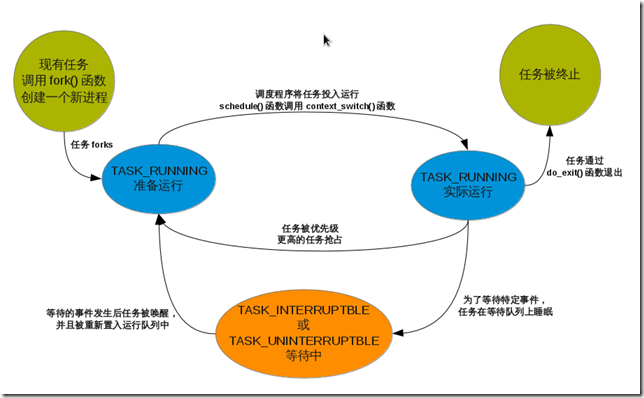
进程标识PID和线程标识TID对于同一个进程或线程来说都是相等的。

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。

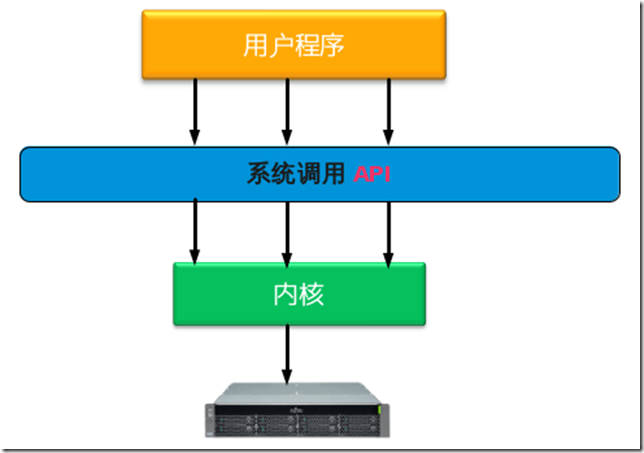
    这样可以避免用户程序不正确的使用硬件设备，从而破坏了其他程序。

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。

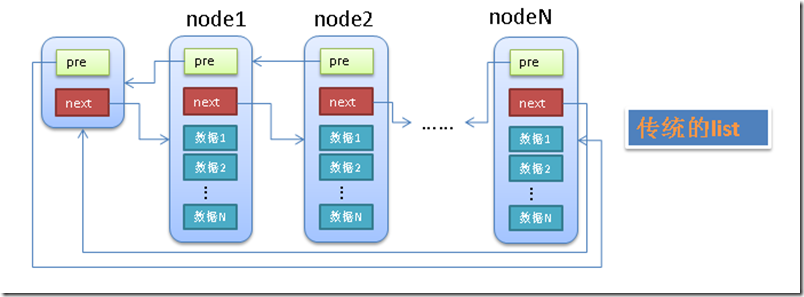
#### 1.2 链表代码的注意点

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：

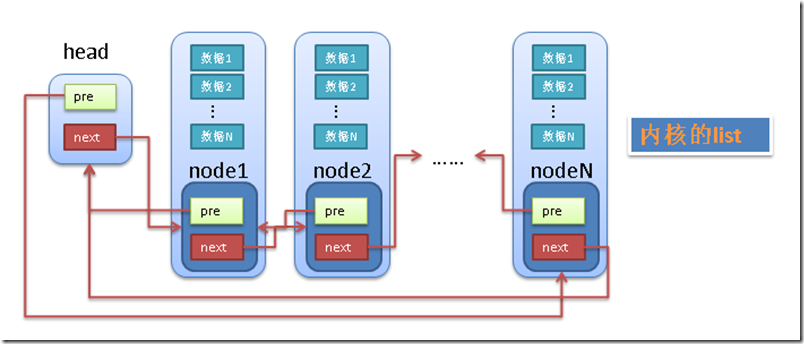
[](https://images0.cnblogs.com/blog/83005/201304/16132037-c83d5c19780d4c13ade4964ed38e6af7.png)

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

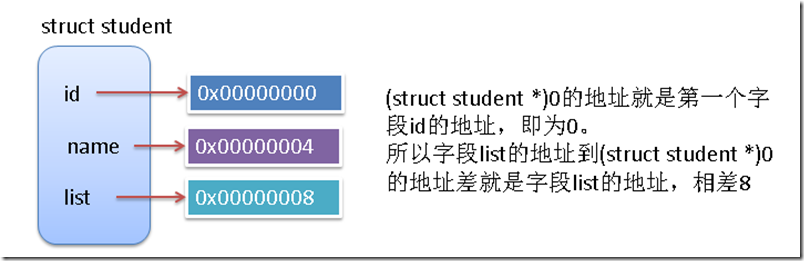
// 其实也就是获取type的地址

[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

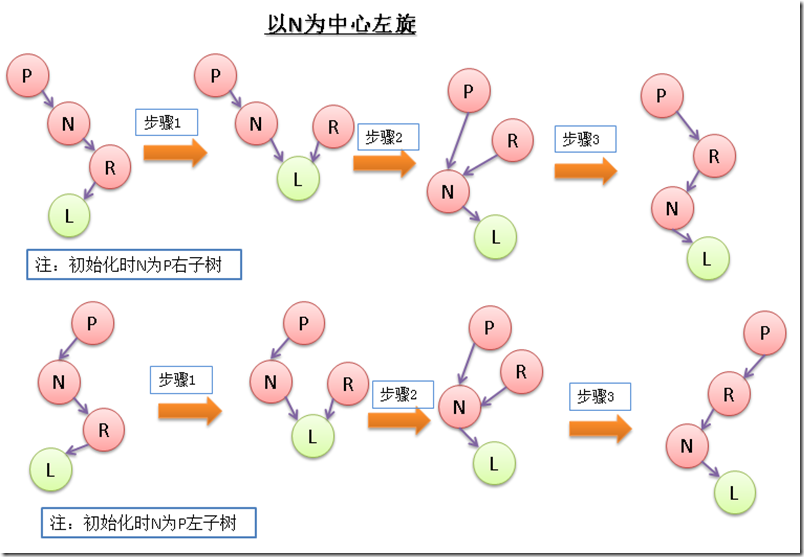
root->rb\_node = right;

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)

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

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

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

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

# [（十二）- 内存管理](http://www.cnblogs.com/wang_yb/archive/2013/05/23/3095907.html)

# [（十三）- 虚拟文件系统](http://www.cnblogs.com/wang_yb/p/3144291.html)

# [（十四）- 块I/O层](http://www.cnblogs.com/wang_yb/p/3299092.html)

# [（十五）- 进程地址空间(kernel 2.6.32.60)](http://www.cnblogs.com/wang_yb/p/3351599.html)

# [（十六）- 页高速缓存和页回写](http://www.cnblogs.com/wang_yb/p/3436126.html)

# [（十七）- 设备与模块](http://www.cnblogs.com/wang_yb/p/3489753.html)

# [（十八）- 内核调试](http://www.cnblogs.com/wang_yb/p/3504539.html)

# [（十九）- 可移植性](http://www.cnblogs.com/wang_yb/p/3512095.html)

# [（二十）- 补丁, 开发和社区](http://www.cnblogs.com/wang_yb/p/3514701.html)

# 参考

https://blog.csdn.net/weixin\_30586085/article/details/101833391