# linux内核分析复习重点

一个脱离了高级趣味的人

打开标题导航，阅读效果最佳。

单纯整理这个复习资料，就整整四天，毕业论文我都没写这么多页。

有不合逻辑或错误的地方，还请批评指正。

## 一、中断和异常

**在X86体系结构下linux内核的中断与异常处理流程？**

### （1）中断、异常描述符表的作用

处理器的速度和外围硬件设备的速度往往不在一个数量级上，内核应该在等待硬件回应期间处理其他事务，等硬件真正完成了请求的操作之后，再对其进行处理。轮询即让内核定期对设备状态进行查询，然后做成相应的处理，但是内核会做很多无用功，因为无论硬件设备是否正在工作，轮询总是周期性地重复执行。中断提供了一种机制，让硬件在需要的时候再向内核发送信号。中断本质上是一种特殊的电信号，处理器接收到中断后，会马上向操作系统反映此信号的到来，然后由操作系统负责处理这些新到来的数据。

中断描述符表（Interrupt Descriptor Table,IDT）是一个系统表，它与每个中断或异常向量相关联，每一个向量在表中有相应的中断或异常处理程序的入口地址。IDT包含三种类型的描述符：任务门、中断门、陷阱门。中断门和陷阱门中都包含段选择符和中断或异常处理程序的段内偏移量。 idtr CPU寄存器使IDT可以位于内存的任何位置，它指定IDT的线性基地址及其限制（最大长度）。

### （2）中断控制器的作用，表述清楚中断如何产生

中断是一种电信号，由硬件设备产生，并通过IRQ输出线送入中断控制器的输入引脚中。中断控制器是个简单的电子芯片，其作用是将多路中断管线，采用复用技术只通过一个和处理器相连接的管线和处理器通信。中断控制器监视IRQ线，检查产生的信号（如果多条IRQ线产生信号，选择引脚编号较小的IRQ线）；当接收到一个中断后，中断控制器会把接收到的引发信号转换成对应的向量，把这个向量存放在一个中断控制器的I/O端口，从而允许CPU通过数据总线读取这个向量；把引发信号发送给处理器的INTR引脚，即产生一个中断；等待，直到CPU把这个中断信号写进可编程中断控制器的一个I/O端口来确认它，此时，清INTR线。

### （3）中断发生时，CPU硬件级的中断信号处理过程

假定内核已被初始化，CPU在保护模式下运行

当执行了一条指令后，cs和eip这对寄存器包含下一条将要执行的指令的逻辑地址.在处理那条指令之前，控制单元会检查在运行前一条指令时是否已经发生了一个中断或异常。如果发生了一个中断或异常，那么控制单元执行下列操作:

1）确定与中断或异常关联的向量i (0 ≤ i ≤ 255)

2）读由idtr寄存器指向的IDT表中的第i项(在下面的描述中，我们假定IDT表项中包含的是一个中断门或一个陷阱门)。

3）从gdtr寄存器获得GDT的基地址，并在GDT中查找，以读取IDT表项中的选择符所标识的段描述符.这个描述符指定中断或异常处理程序所在段的基地址.

4）确信中断是由授权的(中断)发生源发出的。首先将当前特权组CPL(存放在cs寄存器的低两位)与段描述符(存放在GDT中)的描述符特权级DPL比较，如果CPL小于DPL. 就产生一个"Generalprotection" 异常.因为中断处理程序的特权不能低于引起中断的程序的特权.对于编程异常，则做进一步的安全检查:比较CPL与处于IDT中的门描述符的DPL，如果DPL小于CPL，就产生一个"General protection"异常。这最后一个检查可以避免用户应用程序访问特殊的陷阱门或中断门.

5）检查是否发生了特权级的变化，也就是说，CPL是否不同于所选择的段描述符的DPL. 如果是，控制单元必须开始使用与新的特权级相关的栈。通过执行以下步骤来做到这点:

        a.  读tr寄存器，以访问运行进程的TSS段.

        b.  用与新特权级相关的钱段和战指针的正确值装载ss和esp寄存器。这些值可以在TSS中找到

        c.  在新的栈中保存ss和esp以前的值. 这些值定义了与旧特权级相关的梭的逻辑地址.

6）如果故障已发生，用引起异常的指令地址装载cs和elp客存器，从而使得这条指令能再次被执行。

7）在栈中保存eflags、 cs及eip的内容.

8）如果异常产生了一个硬件出错码，则将它保存在栈中.

9）装载cs和eip寄存器，其值分别是IDT表中第i项门描述符的段选择符和偏移量字段.这些值给出了中断或者异常处理程序的第一条指令的逻辑地址。

控制单元所执行的最后一步就是跳转到中断或者异常处理程序.换句话说.处理完中断信号后.控制单元所执行的指令就是被选中处理程序的第一条指令.

中断或异常被处理完后，相应的处理程序必须产生一条iret指令，把控制权转交给被中断的进程，这将迫使控制单元:

 l） 用保存在栈中的值装载cs、 eip或eflags寄存器。如果一个硬件出错码曾被压入校中，并且在eip内容的上面，那么，执行iret指令前必须先弹出这个硬件出错码.

2）检查处理程序的CPL是否等于cs中最低两位的值(这意味着被中断的进程与处理程序运行在同一特权级). 如果是，iret终止执行， 否则，转入下一步.

3）从栈中装载ss和esp寄存器，因此，返回到与旧特权级相关的钱。

4） 检查ds、 es、 fs，&gs段寄存器的内容，如果其中一个寄存器包含的选择符是一个段描述符，并且其DPL值小子CPL，那么，清相应的段寄存器.控制单元这么做是为了禁止用户态的程序(CPL=3)利用内核以前所用的段客存器(DPL=O).如果不清这些寄存器，怀有恶意的用户态程序就可能利用它们采访问内核地址空间.

### （4）系统初始化时，中断与异常初始化过程

内核启用中断以前，必须把IDT表的初始地址装到idtr寄存器，并初始化表中的每项。这项工作是在初始化系统时完成的。

int指令允许用户态进程发出一个中断信号，其值可以是0-255的任意一个向量。因此，为了防止用户通过int指令模拟非法的中断和异常，IDT的初始化必须非常小心。这可以通过把中断或陷阱门描述符的DPL字段设置成0来实现。如果进程试图发出其中的一个中断信号，控制单元将检查出CPL的值与DPL字段有冲突，并且产生一个”Generalprotection”异常。

然而，在少数情况下，用户态进程必须能发出一个编程异常。为此，只要把中断或陷阱门描述符的DPL字段设置成3，即特权级尽可能一样高就足够了。

现在，让我们来看一下Linux是如何实现这种策略的。

中断门、陷阱门及系统门

与在前面”中断描述符表”中所提到的一样，Intel提供了三种类型的中断描述符：任务门、中断门及陷阱门描述符。linux使用与Intel稍有不同的细目分类和术语，把它们如下进行分类：

中断门

用户态的进程不能访问的一个Intel中断门。所有的Linux中断处理程序都通过中断门激活，并全部限制在内核态。

系统门

用户态的进程可以访问的一个Intel陷阱门。通过系统门来激活三个Linux异常处理程序，它们的向量是4，5及128。因此，在用户态下，可以发布into、bound及int$0x80三条汇编语言指令。

系统中断门

能够被用户态进程访问的Intel中断门。与向量3相关的异常处理程序是由系统中断门激活的，因此，在用户态可以使用汇编语言指令int3。

陷阱门

用户态的进程不能访问的一个Intel陷阱门。大部分Linux异常处理程序都通过陷阱门来激活。

任务门

不能被用户态进程访问的Intel任务门。Linux对”Doublefault”异常的处理程序是由任务门激活的。

下列体系结构相关的函数用来IDT中插入门：

set\_intr\_gate(n,addr)

在IDT的第n个表项插入一个中断门。门中的段选择符设置成内核代码的段选择符，偏移量设置成中断处理程序的地址addr，DPL字段设置成0。

set\_system\_gate(n,addr)

在IDT的每n个表项插入一个陷阱门。门中的段选择符设置成内核代码的段选择符，偏移量设置成中断处理程序的地址addr，DPL字段设置成0。

set\_system\_intr\_gate(n,addr)

在IDT的第n个表项插入一个中断门。门中的段选择符设置成内核代码的段选择符，偏移量设置成中断处理程序的地址addr，DPL字段设置成0。

set\_trap\_gate(n,addr)

与前一个函数类似，只不过DPL的字段设置成0。

set\_task\_gate(n,gdt)

在IDT的第n个表项插入一个中断门。门中的段选择符存放一个TSS的全局描述符指针，该TSS中包含要被激活的函数，偏移量设置成0，DPL字段设置成3。

IDT的初步初始化

当计算机还运行在模式时，IDT被初始化并由BIOS例程使用。然而，一旦Linux接管，IDT就被移到RAM的另一个区域，并进行第二次初始化，因为Linux没有利用任何BIOS例程。

IDT存放在idt\_table表中，有256个表项。6字节的idt\_descr变量指定了IDT的大小和它的地址，只有当内核用lidt汇编指令初始化idtr寄存器时才用到这个变量。

在内核初始化过程中，setup\_idt()汇编函数用同一个中断门ignore\_int()来填充所有这256个idt\_table表项。

用汇编语言写成的ignore\_int()中断处理程序，可以看作一个空的处理程序，它执行下列动作：

1. 在栈中保存一些寄存器的内容。
2. 调用printk()函数打印Unknowninterrupt系统消息。
3. 从栈恢复寄存器的内容。
4. 执行iret指令以恢复被中断的程序。

ignore\_int()处理程序应该从不被执行，在控制台或日志文件中出现的Unknowninterrupt消息标志着要么是出现了一个硬件问题，要么就是出现了一个内核的问题。

紧接着这个预初始化，内核将在IDT中进行第二遍初始化，用有意义的陷阱和断处理程序替换这个空处理程序。一旦这个过程完成，对控制单元产生的每个不同的异常，IDT都有一个专门的陷阱或系统门，而对于可编程中断控制器确认的每一个IRQ，IDT都将包含一个专门的中断门。

在接下来的两节中，将分别针对异常和中断来详细地说明这个工作是如何完成。

CPU产生的大部分异常都是由Linux解释为出错条件。当其中一个异常发生时，内核就向引起异常的进程发送一个信号向它通知一个反常条件。这个进程将采取若干必要的步骤来恢复或者中止运行。

异常处理程序有一个标准的结构，由以下三部分组成：

在内核堆栈中保存大多数寄存器的内容。

用高级C函数处理异常。

通过ret\_from\_exception()函数从异常处理程序退出。

为了利用异常，必須对IDT进行适当的初始化，使得每个被确认的异常都有一个异常处理程序。trap\_init()函数的工作是将一些最终值插入到IDT的非屏蔽中断及异常表项中。这是由函数set\_trap\_gate()、set\_intr\_gate()、set\_system\_gate()、set\_system\_intr\_gate()和set\_task\_gate()来完成的。

init\_IRQ()在interrupt数组中找到用于建立中断门的中断处理程序地址，通过set\_intr\_gate()函数将各函数的地址被复制到IDT相应表项的中断门中。

### （5）pt\_regs结构

**pt\_regs结构的作用，结构的相关数据中有的是硬件自动保存，有的是通过程序SAVE\_ALL保存,分别对应的恢复现场方式**



描述了CPU寄存器在内核态的栈中的保存情况。

RESTORE\_ALL和iret。

### 关于中断需要明确：

### 1.init\_IRQ()

**调用init\_IRQ()函数，把中断描述附表的中断处理代码段地址设在在interrupt数组中，该数组指向同一个函数处理common\_interrupt**

#### common\_interrupt的功能

common\_interrupt:

SAVE\_ALL//在栈中保存中断处理程序可能会用到的所有CPU寄存器

//把用户数据段的选择符装到ds和es寄存器

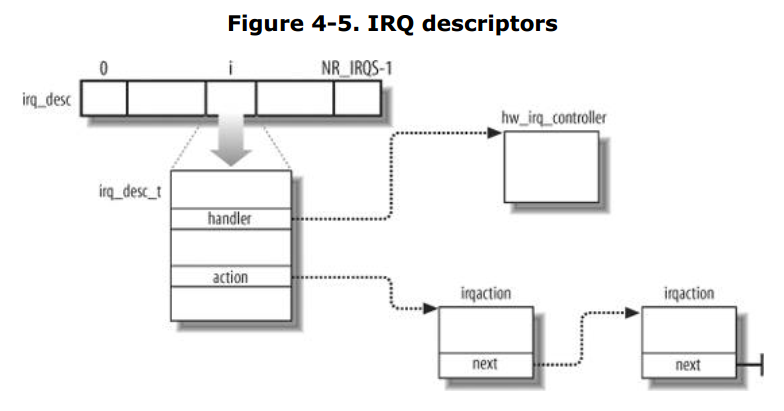
movl %esp,%eax//栈顶地址被保存到eax中

call do\_IRQ //调用do\_IRQ函数执行与一个中断相关的所有中断服务例程

jmp ret\_from\_intr//调度、信号处理、恢复硬件上下文

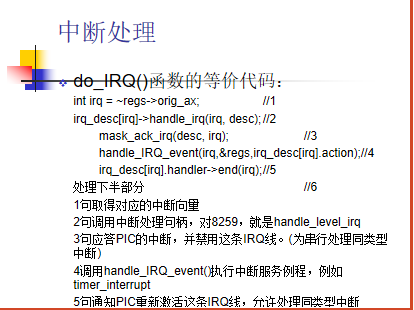
### 2.do\_IRQ

**do\_IRQ函数处理过程，在这里要明确该函数所使用的数据结构irq\_desc数组，讲清楚对irqaction的处理**

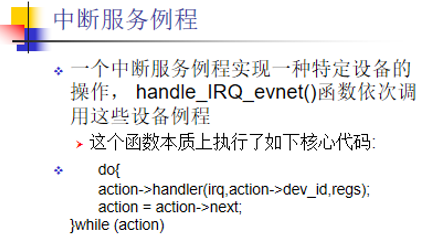


irq\_desc数组中的元素是irq\_desc\_t中断描述符，每一个中断号具有一个描述符，使用action链表连接共享同一个中断号的多个设备和中断。handler points to the hw\_interrupt\_type descriptor that identifies the PIC circuit servicing the IRQ line。handler指向发出这个中断的设备的处理对象hw\_irq\_controller,比如在单CPU，这个对象一般就是处理芯片8259的对象。为什么要指向这个对象呢？因为当发生中断的时候，内核需要对相应的中断进行一些处理，比如屏蔽这个中断等。这个时候需要对中断设备(比如8259芯片)进行操作，于是可以通过这个指针指向的对象进行操作。

irqaction描述符，涉及一个特定的硬件设备和一个特定的中断。handler指向一个I/O设备的中断服务例程(这个handler是irqaction中的),name设备名，dev\_id标志设备本身，next指向描述符链表的下一个元素。



unsigned int do\_IRQ(struct pt\_regs regs)通过pt\_regs结构可以找到中断号，然后对接收到的中断进行应答irq\_desc[irq].handler->ack(irq)，禁止这条线上的中断传递。接下来，do\_IRQ() ensures that a valid handler is registered on the line and that it is enabled and not currently executing.如果是这样的，do\_IRQ就调用handle\_IRQ\_event()来运行为这条中断线所安装的中断处理程序。



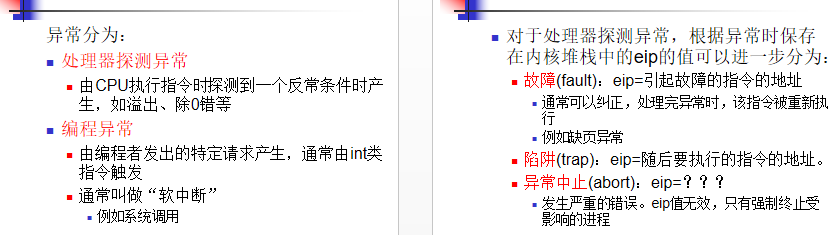
handle\_IRQ\_event(),First,because the processor disabled interrupts,they are turned back on unless IRQ\_DISABLED was specified during the handler’s registration.Next,each potential handler is executed in a loop.Finally,interrupts are again disable and the function returns.

Back in do\_IRQ(),the funcion cleans up and returns to the initial entry point通过irq\_desc[irq].handler->end(irq),which then jumps to ret\_from\_intr().

### 3.用户中断服务例程的注册过程与irqaction对应关系

在激活一个准备利用irq线的设备之前其相应的驱动程序调用request\_irq()注册中断处理程序。这个函数建立一个新的irqaction描述符，并用参数值初始化它。然后调用setup\_irq()函数把这个描述符插入到适合的IRQ链表。如果setup\_irq()返回一个出错码，设备驱动程序中止操作，这意味着IRQ线已有另一个设备所使用，而这个设备不允许中断共享。当设备操作结束时，驱动程序调用free\_irq()函数从IRQ链表中删除这个描述符，并释放相应的内存区。

### 关于异常需要明确：



### 1.异常的初始化

为了利用异常，必須对IDT进行适当的初始化，使得每个被确认的异常都有一个异常处理程序。trap\_init()函数的工作是将一些最终值插入到IDT的非屏蔽中断及异常表项中。这是由函数set\_trap\_gate()、set\_intr\_gate()、set\_system\_gate()、set\_system\_intr\_gate()和set\_task\_gate()来完成的。

### 2.异常处理的主要过程

异常处理有一个标准的结构，由三部分组成

1.在内核态堆栈中保存大多数寄存器的内容

2.调用C语言的函数(异常处理程序)

handler\_name:

pushl $0

pushl $do\_handler\_name

jmp error\_code

3.通过ret\_from\_exception()从异常处理程序退出

### 3.异常与中断的区别

中断——异步的：由硬件设备按照cpu时钟信号随机产生，在程序执行的任何时候可能出现。

异常——同步的：在（特殊的或出错的）指令执行时由CPU控制单元产生，只有在一条指令终止执行后cpu才会发出中断。

异常是由程序的错误产生的，或者是由内核必须处理的异常条件产生的。第一种情况下，内核通过发送一个每个程序员都熟悉的信号来处理异常。第二种情况下，内核执行恢复异常需要的所有步骤。

中断允许内核控制路径嵌套执行，中断处理程序必须永不阻塞，中断处理程序运行期间不能发生进程切换。

假定内核没有bug，大多数异常就只在用户态发生。事实上，异常要么由编程错误引起，要么是由调试程序触发。然而缺页异常发生在内核态。处理缺页异常时，内核可以挂起当前进程，并用另一个进程代替它，直到请求的页可以使用为止。

因为缺页异常处理程序从不进一步引起异常，所以与异常相关的至多两个内核控制路径（第一个由系统调用引起，第二个由缺页引起）会堆叠在一起。

中断处理程序可以抢占其他中断处理程序，也可以抢占异常处理程序。异常处理程序从不抢占中断处理程序。中断处理程序从不执行可以导致缺页的（意味着进程切换）的操作。

### 4.异常的处理过程：重点分析系统调用和缺页异常处理

#### system\_call

1.start\_kernel()：trap\_init()中有一句 set\_system\_trap\_gate(SYSCALL\_VECTOR,&system\_call)。 其中：SYSCALL\_VECTOR：系统调用的中断向量（#define SYSCALL\_VECTOR 0x80），&system\_call汇编代码入口。

2.一执行int $0x80，系统把cs:eip的值存储内核栈中，把ss:esp的值存储内核栈中，把eflags的值存储内核栈中；更新cs:eip的值为0x80的中断服务函数；将ss:esp的值指向了内核栈中的栈顶位置；跳转到system\_call。

3.SAVE\_ALL：保存现场

4.call \*sys\_call\_table(,%eax,4)调用了系统调度处理函数，eax存的是系统调用号，是实际的系统调度程序。 sys\_call\_table：系统调用分派表

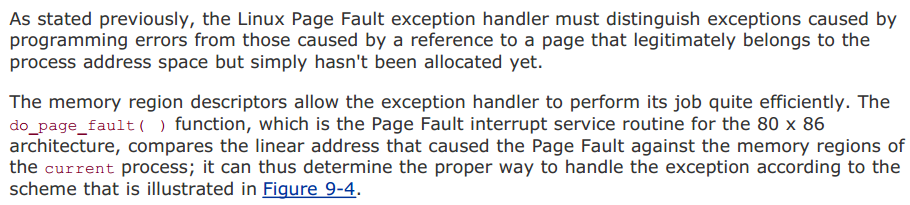
5.syscall\_after\_all：保存返回值

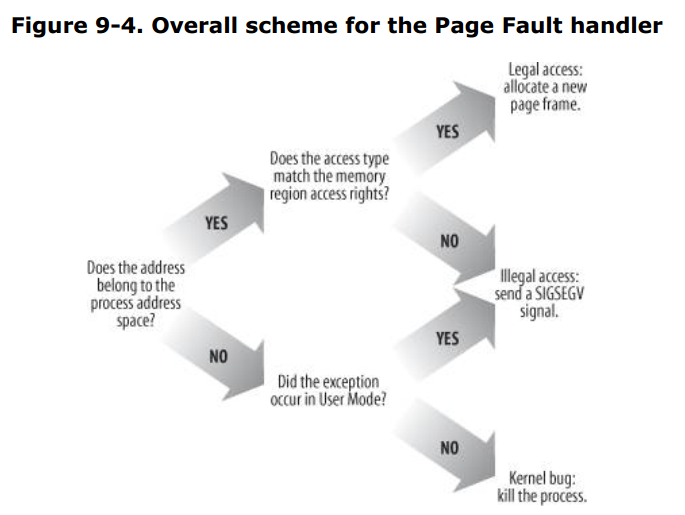
6.syscall退出，若有sys\_exit\_work，则进入sys\_exit\_work:执行的目的是处理进程之间的信号或者进程调度。

7.若无sys\_exit\_work，就执行restore\_all恢复现场。

8.返回用户态，INTERRUPT\_RETURN <=> iret，从内核栈中将:eflags、ss:esp、cs:eip这三个值依次恢复到寄存器中，并换回到用户态中来。

#### do\_page\_fault





## 二、linux中断下半部

**linux中断下半部处理方式，描述软中断及tasklet的处理机制，明确执行时机和处理过程**

### 下半部处理方式

操作系统必须有一个快速、异步、简单的机制负责对硬件做出迅速响应并完成对时间要求严格的操作。中断处理程序很适合实现这些功能，但是对于那些对时间要求宽松，耗时的工作，由于中断处理的局限，适合放在中断处理退出后的某个合适时机执行，这就是中断处理下半部。下半部的任务：就是执行与中断处理密切相关但中断处理程序本身不执行的工作。

下半部实现机制：软中断、tasklets、工作队列。

### 软中断

软件中断(softirqs)：一组静态定义的下半部接口，有32个，可以在所有处理器上同时执行(两个类型相同也可以)，软中断必须在编译期间就进行静态注册，性能非常高。一个软中断不会抢占另外一个软中断，实际上，唯一可以抢占软中断的是中断处理程序。其他软中断(包括同类型)可以在其他处理器上同时执行。

软件中断在编译期间静态分配，用softirq\_action结构表示

struct softirq\_action

{

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

};

Kernel/softirq.c中定义了一个包含32个该结构体的数组

static struct softirq\_action softirq\_vec[NR\_SOFTIRQS] \_\_cacheline\_aligned\_in\_smp; 每注册一个软中断，占据该数组的一项，最多可能32个软中断，目前只用了9个。

### 软中断执行时机

一个注册的软中断必须被标记后才会执行(触发软中断)，通常中断处理程序会在返回前标记它的软中断，使其在稍后被执行。在下列地方，待处理的软中断会被检查和执行：

\*从一个硬件中断代码处返回时

\*在ksoftirqd内核线程中

\*在那些显示检查和执行待处理的软中断代码中，如网络子系统中

### 软中断处理过程

无论用什么办法唤醒，软中断都要在do\_softirq()中执行，该函数循环遍历softirq\_vec[]，被标记的软中断处理函数被逐个执行。

### tasklet

tasklet：tasklet实际上是通过软中断实现的，两个不同类型的tasklet可以在不同的处理器上同时执行，但类型相同的tasklet不能同时时行。tasklet可以动态注册.

tasklet是通过软中断实现的，它由两类软中断代表：HI\_SOFTIRQ和TASKLET\_SOFTIRQ。两者唯一区别是HI\_SOFTIRQ会先于TASKLET\_SOFTIRQ执行。

### tasklet处理过程

Tasklet和高优先级的tasklet分别存放在tasklet\_vec和tasklet\_hi\_vec数组中，数组的每一项针对一个CPU，代表这个CPU上的tasklet链表（这里有点费解，意思是tasklet\_vec和tasklet\_hi\_vec数组中的每个元素都是一个链表），分别由tasklet\_action和tasklet\_hi\_action处理(这两个就是do\_softirq中执行的软中断处理程序)，找到CPU对应的那个项，遍历执行链表上每个待处理的tasklet.

Tasklet在软中断的上下文中运行，所有的代码必须是原子的，不能睡眠、不能使用信号量或其它产生阻塞的函数；

### 工作队列

工作队列在一个内核线程上下文运行，并且可以在延迟一段确定的时间后才执行；有更多的灵活性，它可以使用信号量等能够睡眠的函数。

## 三、linux计时体系结构

**明确主要功能、主要数据结构、主要初始化过程以及时钟中断处理函数注册和调用过程；明确软定时器的组织及处理过程；明确内核对相关时钟源的组织，明确计时时钟源和中断时钟源的区别**

### 主要功能

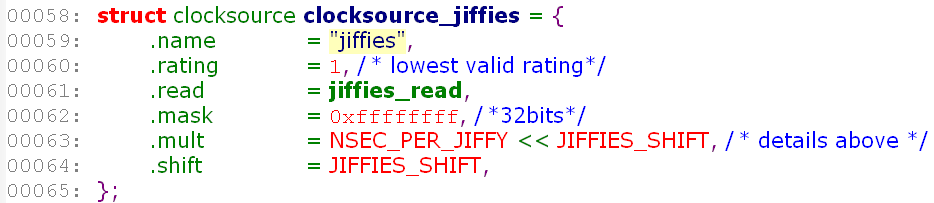
相对于事件驱动而言，内核中有大量的函数都是时间驱动的（时间驱动事件也是事件驱动的一种）。内核需要提供系统定时器、动态定时器、管理系统的运行时间以及当前的日期和时间。

### 主要数据结构

#### struct clocksource

内核用一个struct clocksource结构对真实的时钟源进行软件抽象.

缺省时钟源：



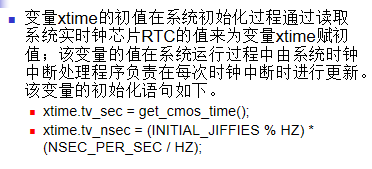
#### jiffies

全局变量extern unsigned long volatile jiffies记录自系统启动以来产生的节拍的总数。每次时钟中断＋1（在系统响应时钟中断，时钟中断处理程序timer\_interrupt()将该变量的值加1 ）。

#### struct timespec xtime

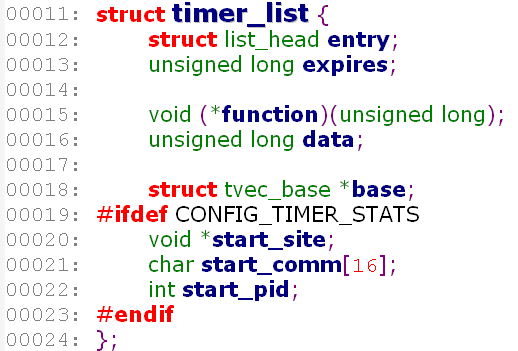
struct timespec xtime当前实际时间，自UTC(1970/1/1)以来经过的时间。

##### xtime初始化和更新





#### 动态定时器timer\_list



系统定时器：周期性产生的事件

动态定时器：推迟到某个确定点执行的事件

### 单处理器系统计时体系结构初始化

在内核初始化期间time\_init()函数被调用来建立计时体系结构，它通常执行如下操作：

1.初始化xtime变量。利用get\_cmos\_time()函数从实时时钟RTC中读取自1970年1月1日（UTC）午夜以来经历的秒数。

2.初始化wall\_to\_monotonic变量。这个变量和xtime一样是timespec类型，只不过它存放将被加到xtime上的秒数和纳秒数，以此来获得单向的时间流。

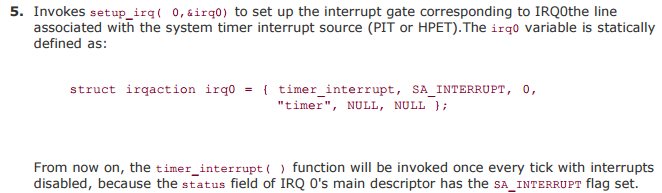
3.如果内核支持HPET，它将调用hpet\_enable()函数确认ACPI固件是否探测到该芯片并将它的寄存器映射到了内存地址空间中。

4.调用select\_timer()来挑选系统中可利用的最好的定时器资源，并设置cur\_timer变量指向该定时器资源对应的定时器对象的地址。

5.调用setup\_irq(0,&irq0)来创建与IRQ0相应的中断门，IRQ0引脚连接着系统时钟中断源（PIT或HPET）

### 中断处理程序注册

计时体系结构的初始化的step 5



### 中断处理函数调用过程

时钟中断处理程序可以划分为两个部分：体系结构相关部分和体系结构无关部分。

与体系结构相关的例程作为系统定时器的中断处理程序time\_interrupt而注册到内核中，以便在产生时钟中断时，它都能够相应地运行。这部分处理程序主要完成：

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

周期性地使用墙上时间更新实时时钟

调用体系结构无关的时钟例程：tick\_periodic()

中断服务程序主要通过调用与体系结构无关的例程tick\_periodic执行下面的工作：

给jiffies\_64变量增加1

更新资源消耗的统计值，比如当前进程所消耗的系统时间和用户时间

执行已经到期的动态定时器

执行scheduler\_tick()函数

更新墙上时间，该时间存放在xtime变量中

计算平均负载

很多重要的操作是在do\_timer和updata\_process\_times中进行。前者承担着对jiffies\_64的实际增加操作，后者更新进程消耗的各种节拍数。

tick\_periodic()执行完毕后，返回与体系结构相关的中断处理程序，继续执行后面的工作，释放xtime\_lock锁，然后退出。

### 动态定时器创建和激活

定时器是一种软件功能，即允许在将来的某个时刻，函数在给定的时间间隔用完时被调用。超时表示与定时器相关的时间间隔已经用完的那个时刻。

内核和进程广泛使用定时器。大多数设备驱动程序利用定时器检测反常情况。

动态定时器（dynamic timer）被动态地创建和撤销，对当前活动动态定时器的个数没有限制。

为了创建并激活一个动态定时器，内核必须：

1.如果需要，创建一个新的timer对象，这可以通过下列方式来进行：

在代码定义一个静态全局变量

在函数定义一个局部变量：在这种情况下，这个对象存放在内核堆栈中。

在动态分配的描述符中包含这个对象。

2.调用init\_timer(&t)函数初始化对象。

3.把定时器到期时激活函数的地址存入function字段。如果需要把参数传递到data字段

4.如果动态定时器还没有被插入到链表中，给expires字段附一个合适的值并调用add\_timer(&t)函数把t元素插入到合适的链表中

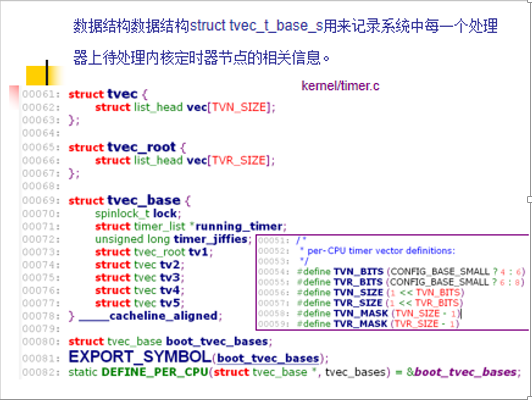
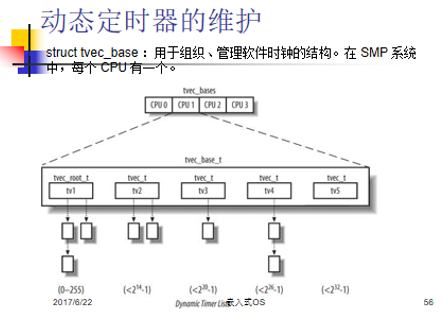
5.否则如果动态定时器已经被插入到链表，则调用mod\_timer()函数来更新expires字段。

一旦定时器到期，内核就自动把元素t从它的链表中删除。不过，有时进程应该用del\_timer()、del\_timer\_sync()或del\_singleshot\_timer\_syn()函数显式地从定时器链表中删除一个定时器。

### 动态定时器维护

把所有定时器放在一个单独的链表中会降低系统的性能，因为在每个时钟节拍去扫描一个定时器的长链表太费时。另一方面，维护一个排序的链表效率也不高，因为插入和删除操作也也非常费时。

解决的办法基于一种巧妙的数据结构，即把expires值划分成不同的大小，并允许动态定时器从大expires值的链表效率到小expires值的链表进行有效的过滤。此外，在多处理器系统中活动的动态定时器集合被分配到各个不同的CPU中。



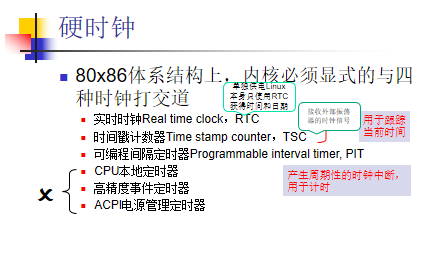
### 动态定时器处理

尽管软定时器具有巧妙的数据结构，但是对其处理是一种耗时的活动，所以不应该被时钟中断处理程序执行。在Linux2.6中该活动由延迟函数来执行，也就是由TIMER\_SOFTIRQ软中断执行。

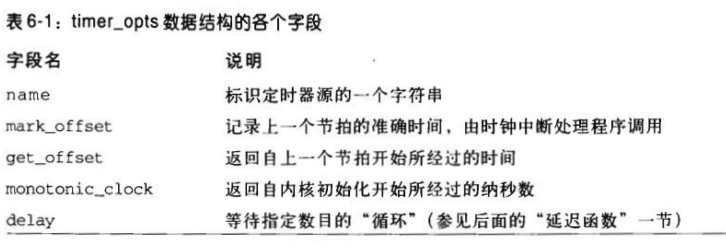
run\_timer\_softirq()函数是与TIMER\_SOFTIRQ软中断请求相关的可延迟函数，找出所有到期的软件时钟，然后依次执行其处理函数。

### 内核对相关时钟源的组织

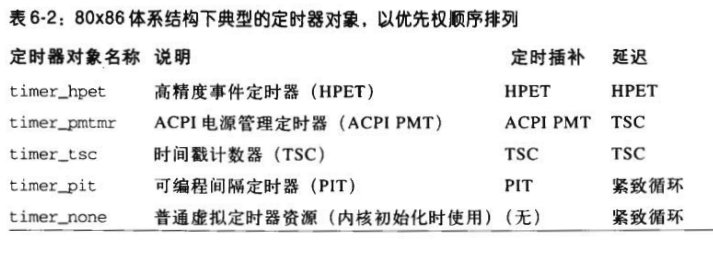
这个问题我没弄明白时钟源和定时器什么区别。尽量写吧。



定时器对象timer\_opts.包含名称和四个标准方法。



定时器对象中最重要的方法是mark\_offset和get\_offset.mark\_offset方法由时间中断处理程序调用，并以适当的数据结构记录每个节拍到来时的准确时间。get\_offset方法使用已记录的值来计算自上一次时钟中断(节拍)以来经过的时间(us为单位).由于这两种方法，使得Linux计时体系结构能够达到子节拍的分辨度，也就是说，内核能够以比节拍周期更高的精度来测定当前的时间，这种操作被称作“定时插补(timerinterpolation)”



变量cur\_timer存放了某个定时器对象的地址，该定时器是系统可利用的定时器资源中最好的。

clock source用于为linux内核提供一个时间基线，如果你用linux的date命令获取当前时间，内核会读取当前的clock source，转换并返回合适的时间单位给用户空间。在硬件层，它通常实现为一个由固定时钟频率驱动的计数器，计数器只能单调地增加，直到溢出为止。时钟源是内核计时的基础，系统启动时，内核通过硬件RTC获得当前时间，在这以后，在大多数情况下，内核通过选定的时钟源更新实时时间信息（墙上时间），而不再读取RTC的时间。

### 计时时钟源和中断时钟源

这个问题我没看懂，我的理解是：

首先时钟源是重心词，计时和中断都是定语。

计时时钟源是保存当前时间和日期的，对应的是时间戳计数器。

中断时钟源是基于时钟中断信号（告诉内核某一段时间间隔过去了）实现的时钟源，例如jiffies时钟源。一般计时时钟源精度高于中断时钟源。

## 四、模块

**模块的编写、编译；另外给内核源码直接添加新的驱动模块代码，并且要求在编译内核是能够看到并配置，需要做哪些改动？**

### 模块的编写

一个典型的内核模块包含：

头文件声明：module.h和init.h是必须的。

模块许可声明

初始化、清理函数的定义

module\_init和module\_exit注册初始化和清理函数

### 模块的编译

采用kbuild构建系统，kbuild makefile一个主要的功能是指定编译什么，通过obj-?和xxx-objs指定。

obj-?指定编译什么，怎么编译。?取值y或者m(编译进内核或编译成模块)。obj-?=$(target).o,如果未指定$(target)-objs则所需源文件为$(target).c或者$(target).s.如果指定了$(target)-objs，则编译需要的源文件由其指定，且不能有$(target).c或者$(target).s

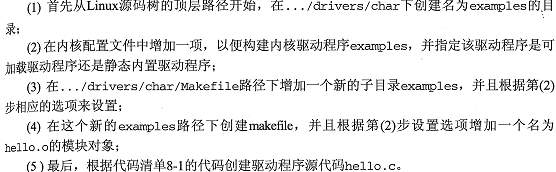
在linux2.6内核中，模块的编译需要配置过的内核源代码，没有源代码无法进行内核模块编译工作。编译、链接后生成的内核模块后缀为.ko。

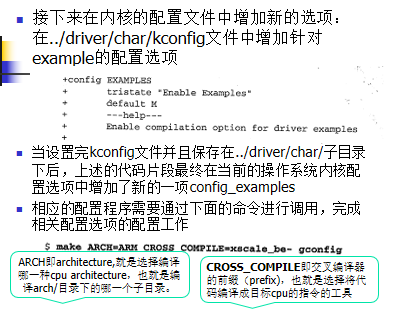
编译过程中首先会到内核源码目录下，读取顶层的makefile文件，然后返回模块源代码所在的目录编译。

### 模块的加载

编译好模块后用户可以利用超级用户的身份可以将内核模块加载到内核中。常见的实用程序insmod、rmmod、lsmod、modprobe。

直接在内核源码树下完成模块的构建





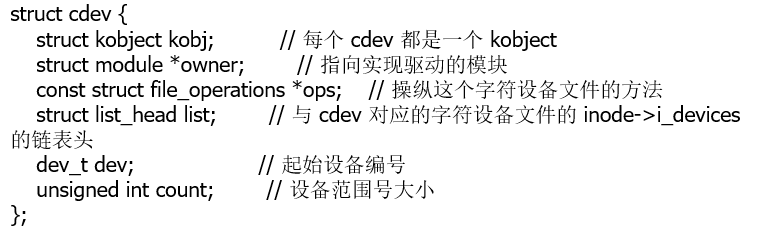
## 五、字符设备驱动程序

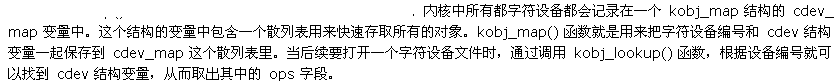
**字符设备驱动程序的组成及linux对字符设备的管理及访问机制**

**一定需要明确cdev结构、file\_operation、设备文件及其inode、设备号、内核管理字符设备数组、并且要搞清楚字符设备的访问过程一定要和文件系统相结合**

### cdev结构

字符设备驱动程序是由一个cdev结构描述的。



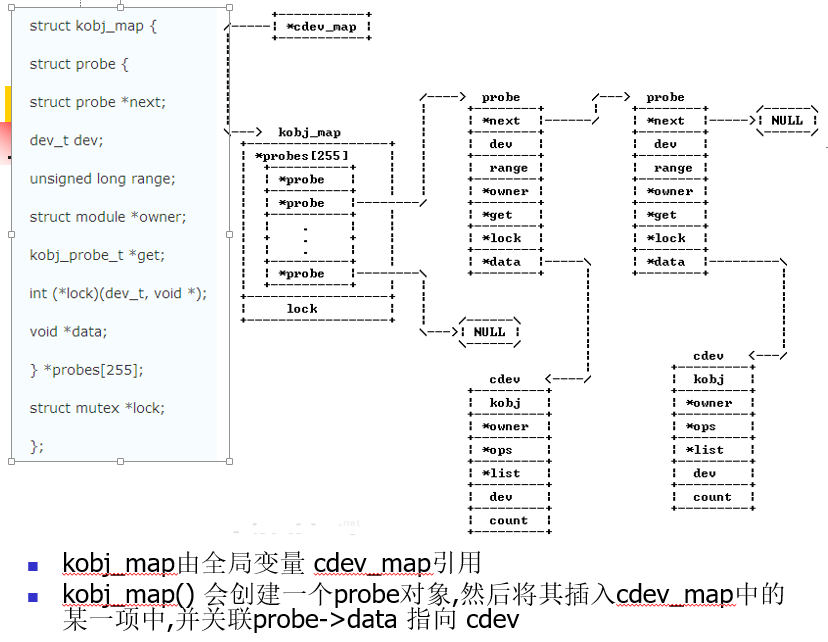


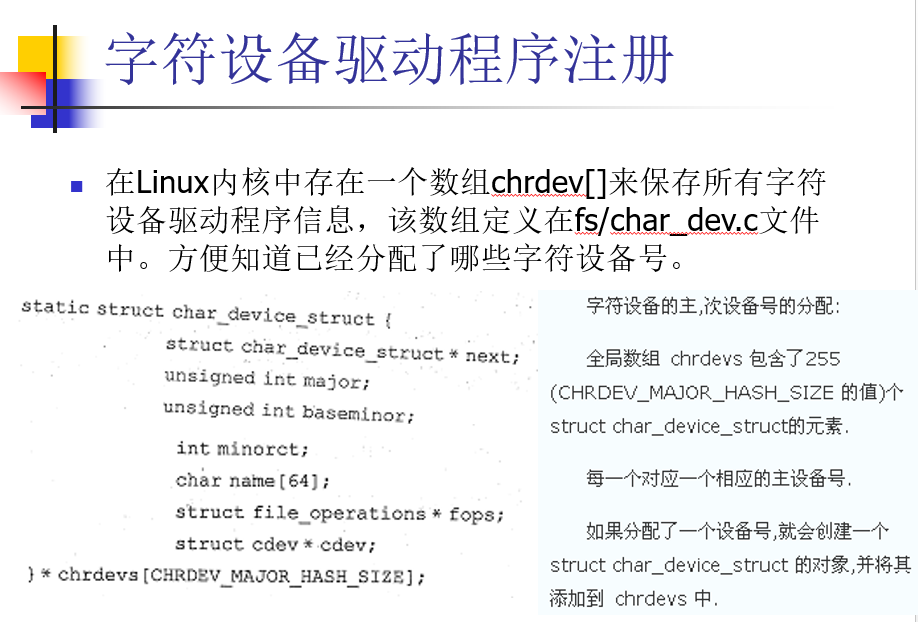
中文版翻译好多费解的地方，这里抄了一段英文。中文请看深入linux内核551页。

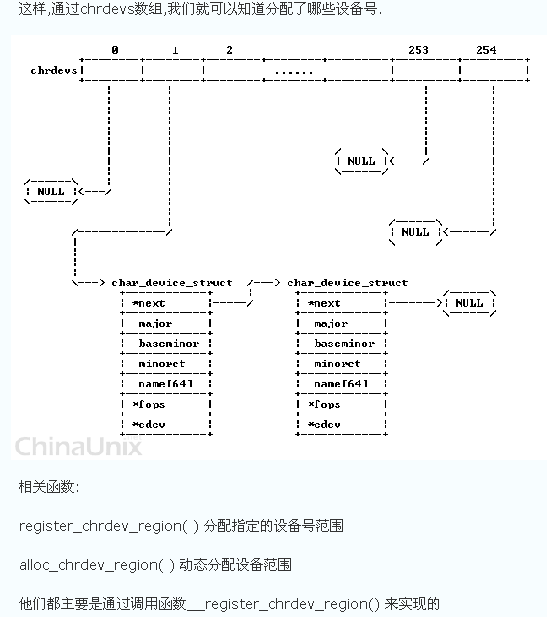
The device driver model defines a kobject mapping domain for the character devices,which is represented by a descriptor of type kobj\_map and is referenced by the cdev\_map global variable. The kobj\_map descriptor includes a hash table of 255 entries indexed by the major number of the intervals. The hash table stores objects of type probe,one of each registered range of major and minor numbers.

When the kobj\_map() （不要和上一段中的a descriptor of type kobj\_map混淆，这个函数在cdev\_add中被调用，作用是把设备号范围复制到设备驱动程序的描述符中）function is involked,the specified interval of device numbers is add to the hash table. The data field of the corresponding probe object points to the cdev descriptor of the device driver.

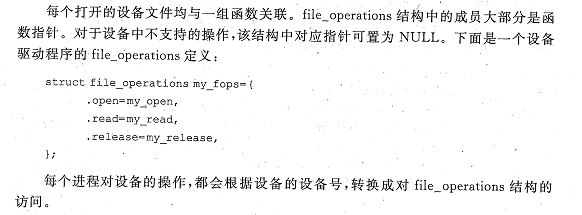
The kobj\_lookup() function receives as input parameters a kobject mapping domain and a device number; it searches the hash table and returns the address of the kobject of the ower of the interval including the number, if it was found. When applied to the mapping domain of the character devices,the function returns the address of the kobject embedded in the cdev descriptor of the device driver that owns the interval of device numbers.



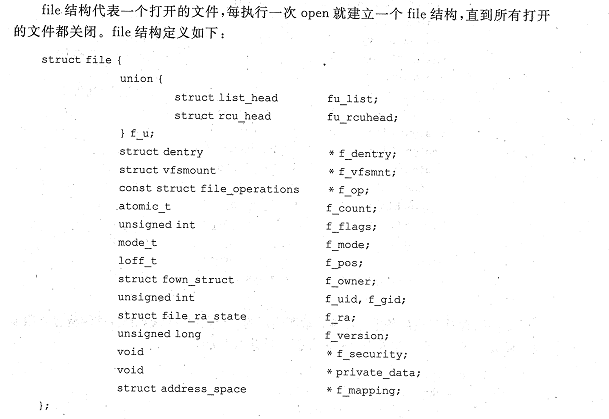




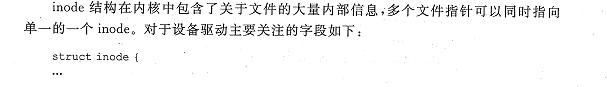
### file\_operations

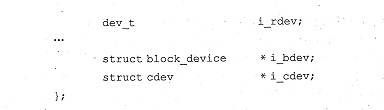


### file



### inode





### 设备号dev\_t

Linux系统使用设备号来标识设备文件。设备号分为主设备号和从设备号。

主设备号是同一类设备的标识，对应着一个驱动程序

从设备号是在驱动程序中来指示某个物理设备的实例，从设备号使得不同的物理实例可以使用同一个驱动程序。

### 访问字符设备



## 六、VFS文件系统

**VFS文件系统的作用、主要数据结构、与进程有关的文件系统的数据结构及作用？VFS文件系统如何实现与具体文件系统的兼容？相关具体文件系统的VFS超级块对象何时被创建？如何被创建？文件系统被挂载时该文件系统必须是注册过的原因？系统描述open系统调用的执行及对文件的访问过程？与进程有关的文件系统的数据结构要仔细理解。**

### 虚拟文件系统作用

为用户空间程序提供了文件和文件系统相关的接口。系统中所有文件系统不但依赖vfs共存，而且也依靠vfs系统协同工作。程序可以利用标准的unix系统调用对不同的文件系统，甚至不同介质上的文件系统进行读写操作。

### 主要数据结构

1. 超级块对象super\_block：保存文件系统信息
2. 索引结点对象inode：存放具体文件的一般信息。
3. 目录项对象dentry：存放目录项和文件链接的信息。
4. 文件对象file：保存已打开的文件和进程的交互信息。

在磁盘中的数据结构：

* 1. 引导控制块：操作系统的初始引导块，用以读入并启动操作系统。一个文件系统，只有根文件系统才含有引导控制块。
  2. 盘控制块：包含整个逻辑盘的资源管理信息，如逻辑盘中的块数、块大小、空闲块数、空闲块地址、空闲FCB数以及FCB地址。通常被称为超级块。
  3. 文件控制块：包含文件的属性信息，如文件访问权限、大小、存储地址。被称为i节点(inode)

这里好多同学会认为盘控制块对应的就是super\_block,文件控制块对应的是imode。其实不是的，super\_block是对所有文件系统的盘控制块的抽象，super\_block中有个字段void \*s\_fs\_info指向的就是文件系统的特殊信息。 同理inode亦然。super\_block和inode都是文件系统从磁盘中提取信息填充的。对比ext2文件系统的ext2\_super\_block和ext2\_inode就可理解这些。

在内存中的数据结构：

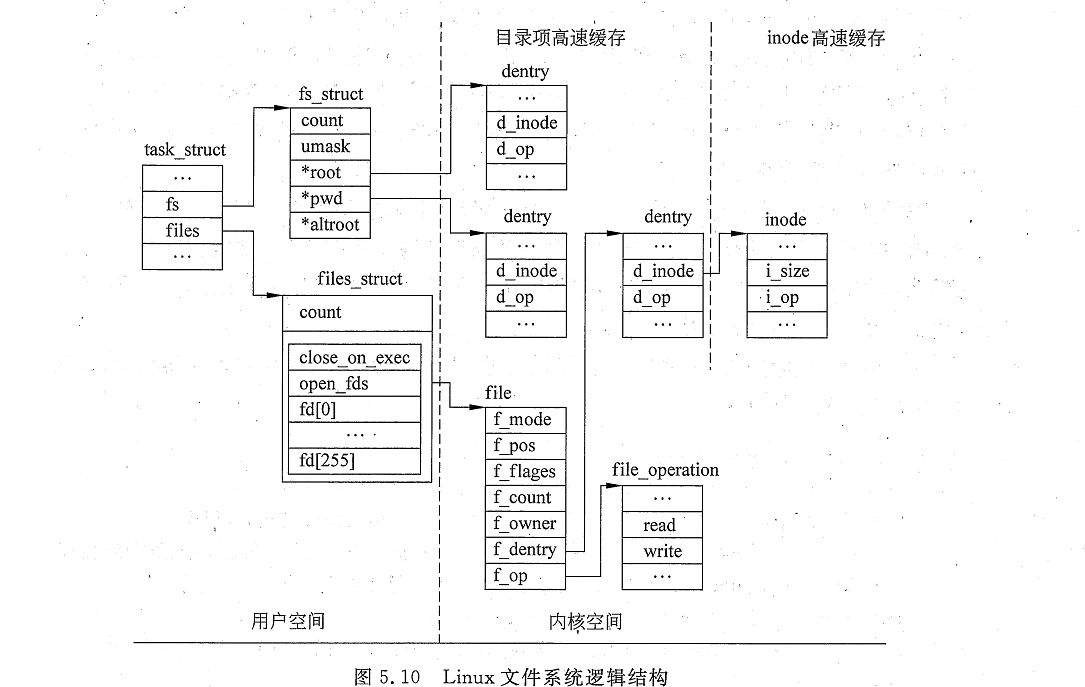
* 1. 系统打开文件表：包含每个已打开文件的FCB的副本，记录打开该文件的进程数，以及其他信息
  2. 进程打开文件表：包含一个指向系统打开文件表相应项的指针，以及其他信息

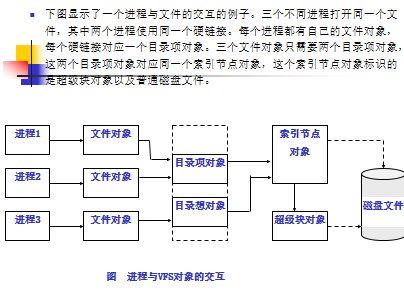
这里内存中的数据结构，我不清楚为什么李老师会这样区分，磁盘中的数据结构确实是上述中的三个，内存中的数据结构多了去了。

### 与进程有关的文件系统的数据结构

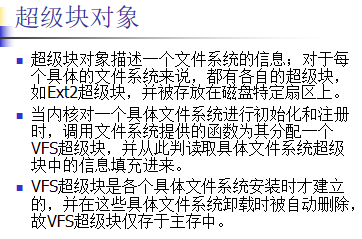
用户打开文件相关信息files\_struct

进程的当前的工作目录和根目录的相关信息fs\_struct



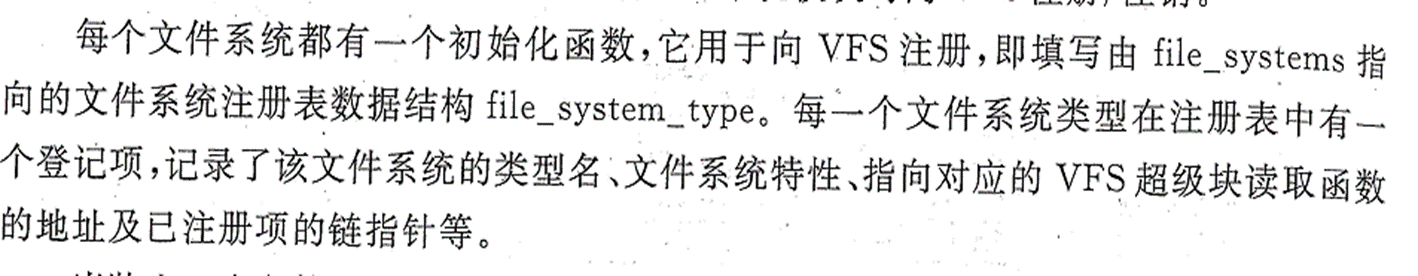


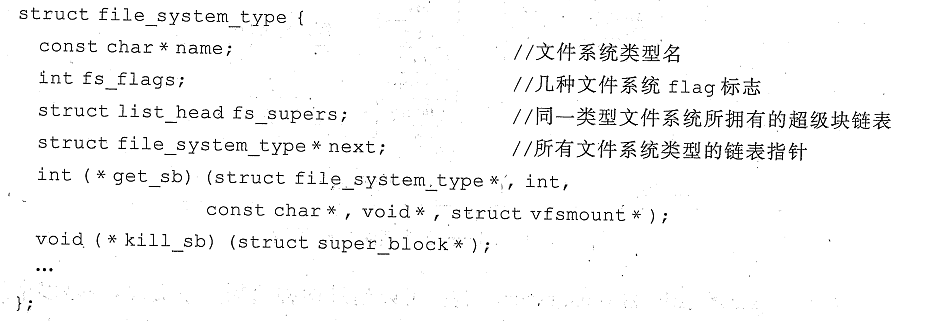
### 相关具体文件系统的VFS超级块对象的创建

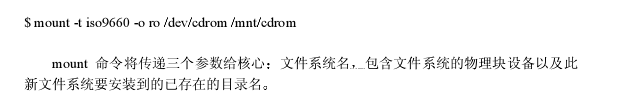


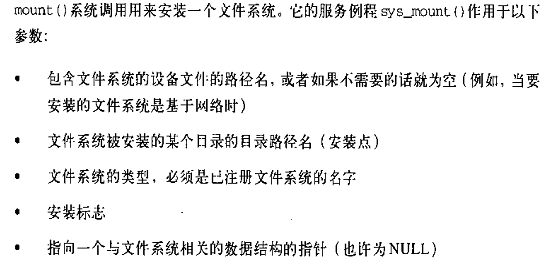
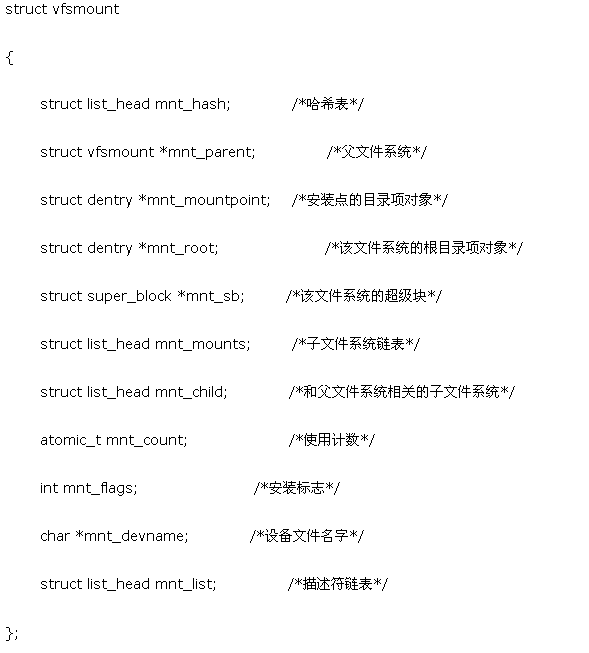
### 安装普通文件系统

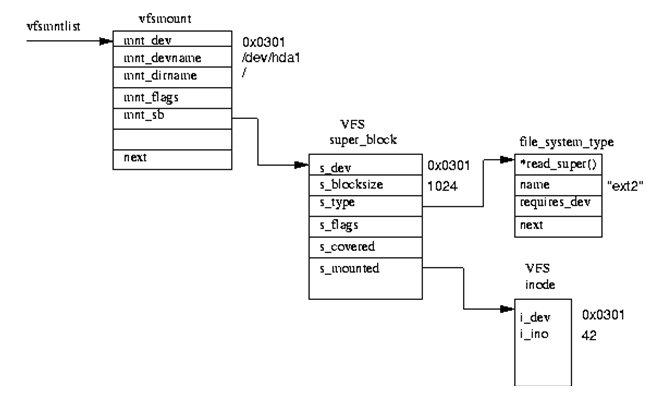
注册文件系统file\_system\_type用来描述各种特定文件系统。







当文件系统被实际安装，将有一个vfsmount结构体在安装点被创建。  




vfsmount 将“安装点”dentry 和“被安装设备”的根目录节点 dentry 关联起来。

所以，在安装文件系统时，内核的主要工作就是：

1、 创建一个 vfsmount

2、 为“被安装设备”创建一个 super\_block，并由具体的文件系统来设置这个 super\_block。

3、 为被安装设备的根目录节点创建 dentry

4、 为被安装设备的根目录节点创建 inode， 并由 super\_operations->read\_inode() 来设置此 inode

5、 将 super\_block 与“被安装设备“根目录节点 dentry 关联起来

6、 将 vfsmount 与“被安装设备”的根目录节点 dentry 关联起来

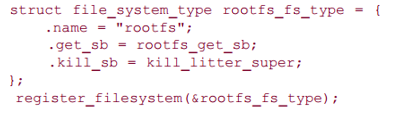
### 根文件系统的挂载

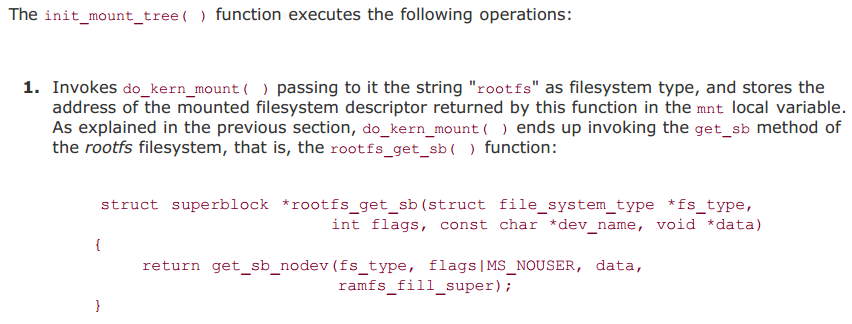
安装根文件系统

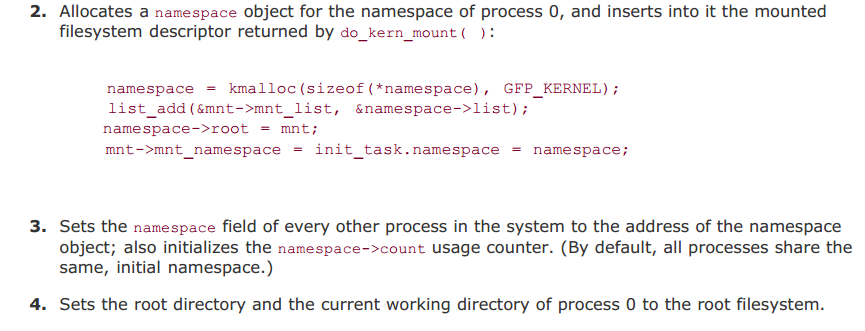
第一阶段：安装一个特殊rootfs文件系统，该文件系统仅提供一个作为初始安装点的空目录

该阶段有init\_rootfs()和init\_mount\_tree()函数完成，它们在系统的初始化过程中执行。

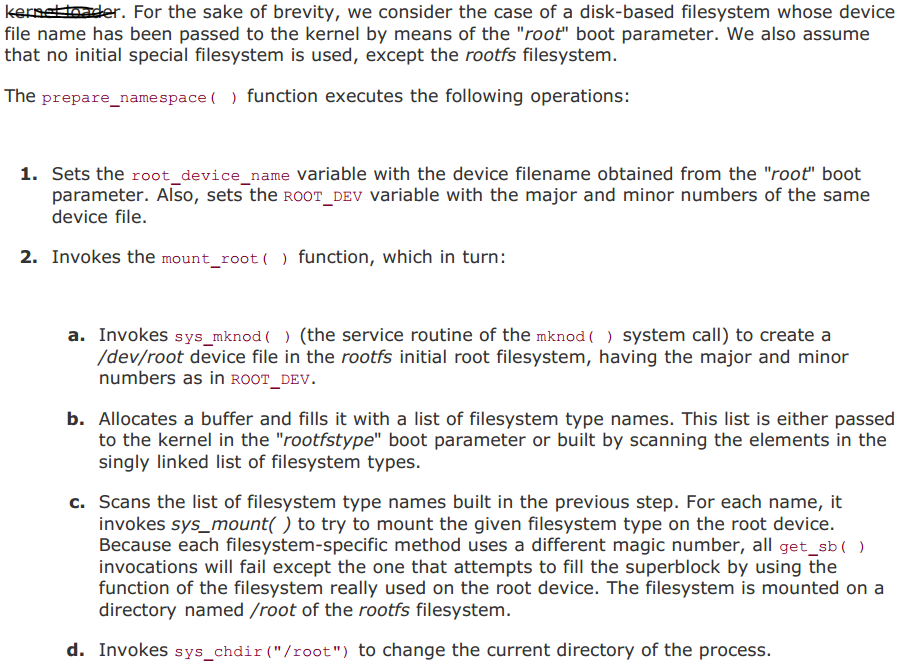
init\_rootfs()函数注册特殊文件系统类型rootfs:

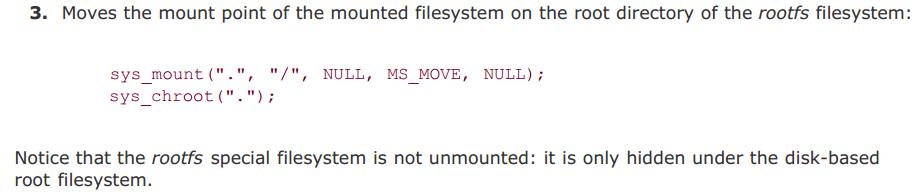






第二阶段：内核在空目录上安装实际根文件系统





### open系统调用的执行及对文件的访问过程

<http://blog.chinaunix.net/uid-28362602-id-3425578.html>

open在内核里面的入口函数时sys\_open

sys\_open函数内容

do\_sys\_open(AT\_FDCWD, filename, flags, mode)

1.找到一个本进程没有使用的文件描述符fd（int型）

在当前进程打开的文件位图表中，找到第一个为0的位，返回这个位在位图表里的下标

2.分配一个全新的struct file结构体

把当前进程的文件表扩展一个文件（即尝试添加一个struct file到当前进程的文件列表中），进程task\_struct-> files\_struct-> fd\_array[NR\_OPEN\_DEFAULT]是一个struct file 数组

3.根据传入的pathname查找或建立对应的dentry

1).如果pathname中第一个字符是“/”，那么说明使用绝对路径，设置nameidata为根目录对应的dentry和当前目录的inode，mount点等

2).如果不是“/”，则使用相对路径，设置nameidata为当前目录对应的dentry，inode，mount点等

3).一层一层往下查找，直到找到最终的那个文件或者目录分量，注意：如”/usr/bin/make”，先找“/”（这是3.1就做了的），再找“/”下面的usr,再找bin，最后找make。

这里细说一下第一层怎么在“/“下面找到”usr“的：

第一层查找先找“/”下面的usr对应的dentry，内核通过“/”对应的dentry和”usr”字符串两个参数进行hash运算获取一个dentry的链表。然后逐个看这个链表里面有没有parent dentry为“/”对应的dentry的，以及dentry对应的名字的hash值是否与“usr”对应的hash值相同。

前面条件都满足这里再看一下parent dentry是否有DCACHE\_OP\_COMPARE标识，如果有此标识且文件系统实现了dentry->dentry\_operations->d\_compare函数，那么就调用文件系统的这个函数来判断。

如果条件都符合，那么说明内存中usr对应的这个dentry是存在的，如果这个dentry->d\_flags中包含DCACHE\_OP\_REVALIDATE，那么就会调用此dentry->dentry\_operatoin->d\_revalidate来进行一次核对（网络文件系统此函数都实现了，以便于远程的便跟，在这里会得到更新）。

如果最终usr对应的dentry不存在，那么内核就在内存中直接分配一个dentry结构体并且把dentry的name和“usr”对应起来，并且设置这个dentry的parent为“/”对应的dentry，然后还要调用”/”对应的dentry->d\_inode这个inode的inode\_operation->lookup(“/”的inode，新建的dentry，flags)，如果返回了新的dentry，那么就把dentry结构体指针指向新返回的dentry，否则还是返回刚刚新创建的那个dentry。（一般的文件系统都实现了inode\_operation->lookup，我猜他们会在这个函数里面如果/usr存在则把dentry对应的inode给设置好。如果/usr不存在则返回一个NULL之类的，以一个错误跳出整个路径执行）。

到这里，无论是dentry已经存在于内存中找到的，还是新创建的dentry，总之，对应于“usr”结构的dentry在内存中已经存在了。然后调用follow\_mount()函数找到“usr”最新的vfsmount（这里有一点点麻烦，后续会专门讲，这里只需要指定如果”/dev/sda” mount 到了/mnt,/dev/sdb 也mount到了/mnt,那么这里返回的是最新的这条/dev/sdb mount到/mnt这个vfsmount）。

然后把这个已经找到的或创建的dentry（已经存在于内存中的dentry已经有了inode和它绑定，新建立的dentry也通过inode\_operation->lookup建立起来了inode和dentry的联系(此函数会在操作真正的磁盘介质吧inode读出来)）和这个最新的vfsmount存到struct path中。

然后把这个含有dentry，vfsmount的path结构体存入nameidata数据结构中，到这里，“usr“对应的dentry，inode，vfsmount都准备好了，且存到了nameidata中了

4).接着3)里面，一层一层的往下找，依次会找到usr，bin，最后到了”make”

这里就不调用一层一层往下找的函数了，进入另外一个函数do\_last（）函数来处理。在dolast，在dolast里面如果此dentry不存在则创建它，如果有O\_CREATE标识则创建这个文件的inode（这里会调用vfs\_create函数，继续调用dentry->inode\_operation->create来建立inode，文件系统实现的此函数会操作正在的磁盘介质去创建inode），并且建立inode和dentry的联系，并且建立”make”对应的vfsmount为最新的mount结构，至此，“/usr/bin/make”中最后一个分量“make”的dentry，inode，vfsmount都存到nameidata中去了。

接着还会把2中分配的file结构体的path(包含dentry和vfsmount)的dentry分量设置为nameidata的这个dentry（dentry结构体中已经有inode的指针），vfsmount也设置为nameidata的vfsmount，并且设置file结构体的file->f\_mapppin为nameidata中dentry的inode的i\_mapping.并且设置file->f\_pos指针为0。

至此，make对应的新分配的这个struct file结构体中的dentry，inode，vfsmount都为nameidata中的了，并且struct file映射到内存的address\_space也设置为了inode对应的address\_space，struct file的当前位置指针设置为了0，“make”分量的这个struct file结构体准备好了。

接着还会把这个struct file结构体加入其inode对应的super\_block超级块的s\_files链表中，即struct file结构体会加入其自身inode所在超级块的所有文件链表中。

并且如果自身inode的file\_operations不为空则还会设置这个struct file的file\_operation等于inode的这个file\_operations，即公用inode的file的操作方法。如果inode的file\_operations没有实现，则设置为空。

设置此文件标识符为FILE\_OPENED.

4.建立fd到这个struct file结构体的联系

调用fd\_install(fd,f)来把1中分配的文件描述符和2中的struct file建立联系。

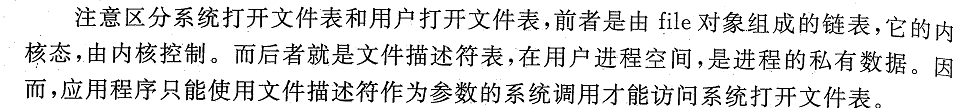
先获取当前进程的fdtable（简单可理解为进程的关联的所有文件数组）的所有文件数组fdtable=current->files->fdt，（current为当前进程task\_struct），设置fdtable->fd[fd]=file，（下标fd即新分配的文件描述符，file即为3中创建的struct file结构体）。

这样，进程和文件描述符，struct file，dentry，inode，vfsmount就全部关联起来了。

<https://mp.weixin.qq.com/s?__biz=MzI3NzA5MzUxNA==&mid=2664602545&idx=1&sn=420236c7883dac2b5bdd6f720d1ccb74&scene=0&utm_source=tuicool&utm_medium=referral>

这个也写的很好，多读几次，对理解文件系统和模块有很多帮助。

### 用户打开文件表和系统打开文件表



file结构除保存文件当前位置外，还把指向该文件inode的指针放在其中，并形成一个双向链表，称系统打开文件表。（这李老师从哪里抄的这一句，这真是瞎写，file中不存在inode，存在一个dentry,这个dentry中有个inode倒是真的。）

文件对象通过一个名为filp的slab高速缓存分配，filp描述符地址存放在filp\_cachep变量中。

文件系统对象包含在由具体文件系统的超级块所确立的几个链表中。每个超级块对象把文件对象链表的头存放在s\_files字段中；因此，属于不同文件系统的文件对象就包含在不同的链表中。链表中分别指向前一个元素和后一个元素的指针都存放在文件对象file的f\_list字段中。files\_lock自旋锁保护超级块的s\_files链表免受多处理器系统上的同时访问。

文件对象的f\_count字段是一个引用计数器：它记录使用文件对象的进程数（记住，以CLONE\_FILES标志创建的轻量级进程共享打开文件表，因此他们可以使用相同的文件对象）。当内核本身使用该文件对象时也要增加计数器的值——例如，把对象插入链表中或发出dup()系统调用时。

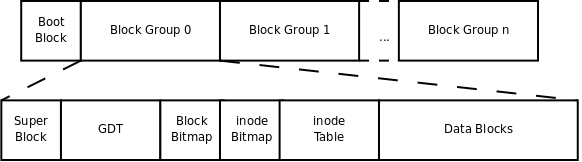
当VFS代表进程必须打开一个文件时，它调用get\_empty\_filp()函数来分配一个新的文件对象。该函数调用kmem\_cache\_alloc()从filp高速缓存中获得一个空闲的文件对象，然后初始化这个对象的字段。

用户打开文件相关信息files\_struct

进程的当前的工作目录和根目录的相关信息fs\_struct

## 七、ext2文件系统

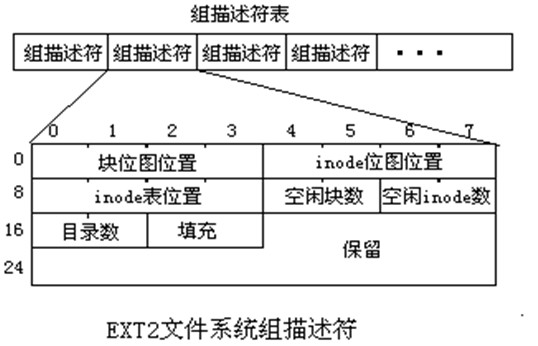
**ext2文件系统磁盘的数据结构；EXT2文件系统中关于索引节点中的i\_block[]的组织及数据的寻址？文件空洞产生原因与处理？PPT中关于上述内容要注意掌握。**

****

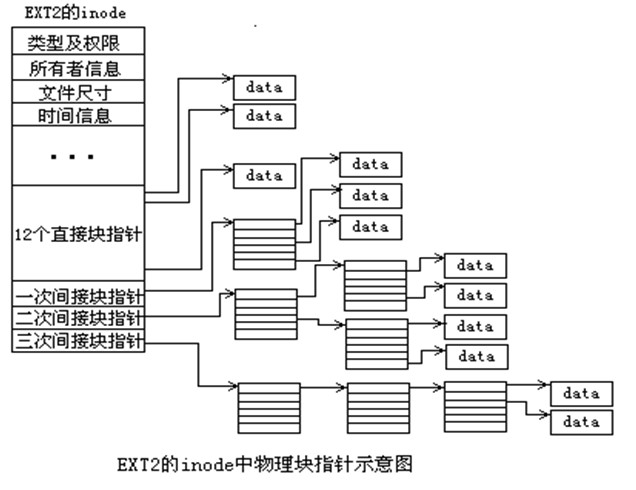
### ext2\_super\_block

****

### ext2\_group\_desc

****

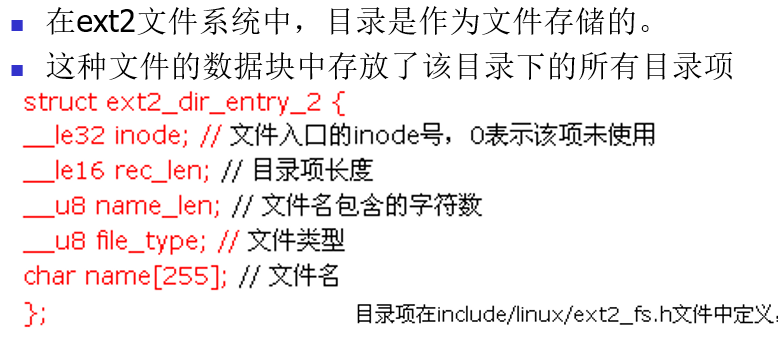
### ext2\_inode

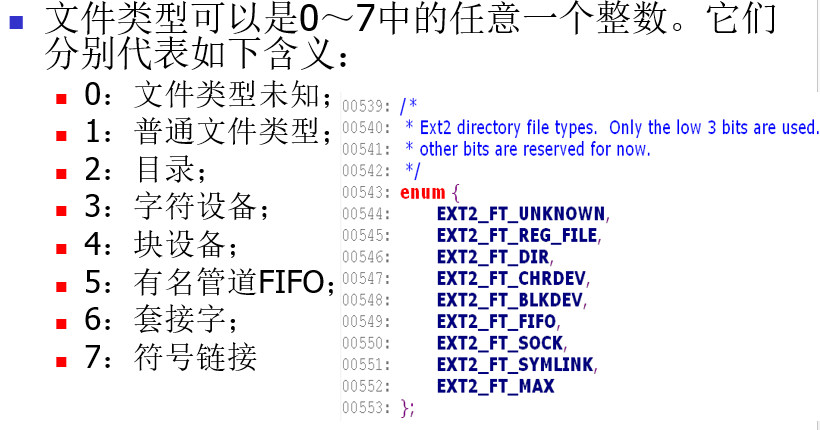
****

#### 索引节点中的i\_block[]

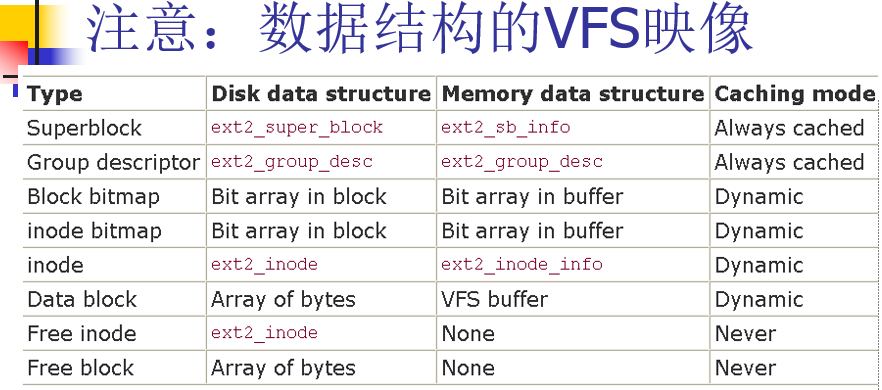


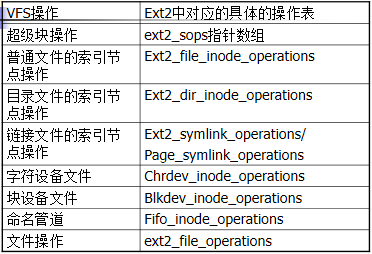
### ext2\_dir\_entry\_2



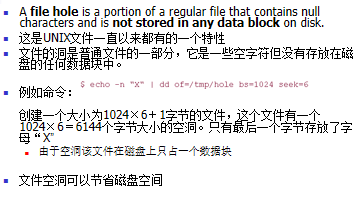


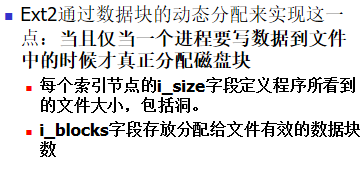
### Ext2数据结构的vfs映像

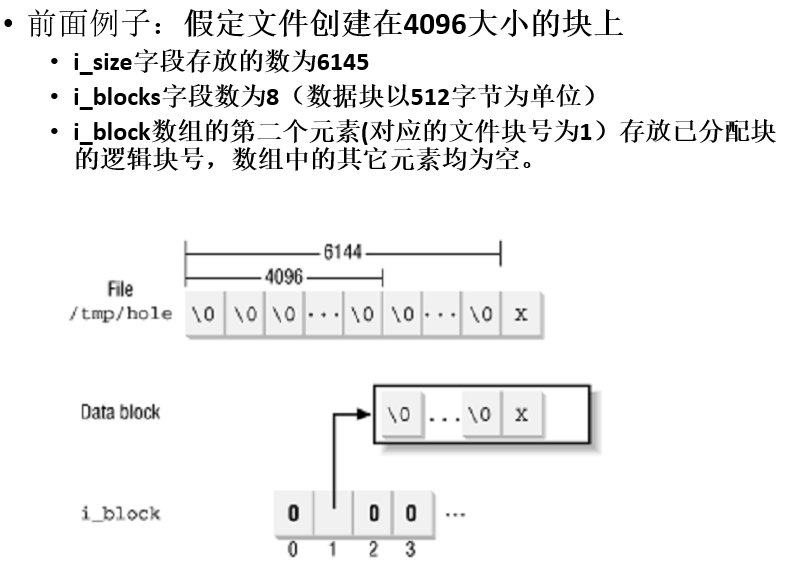




### 文件空洞







## 八、内存寻址

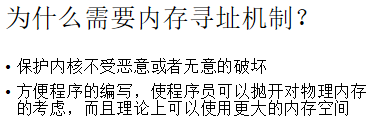
**何为逻辑地址、线性地址及物理地址他们的相互关系？在linux中逻辑地址与线性地址的关系及原因？关于对页表项的理解？进程地址空间与进程页表的关系？弄清楚Linux如何有效地利用x86的分段和分页机制把逻辑地址转换为物理地址。**

### 逻辑地址、线性地址及物理地址

逻辑地址：包含在机器语言指令中用来指定一个操作数或一条指令的地址。在i386体系中每一个逻辑地址都有一个段标识符（segment，又称段选择符，是个索引值 16位）和偏移量（offset，32位）组成。

线性地址：一个32位的无符号整数，可以用来表示高达4GB的地址。

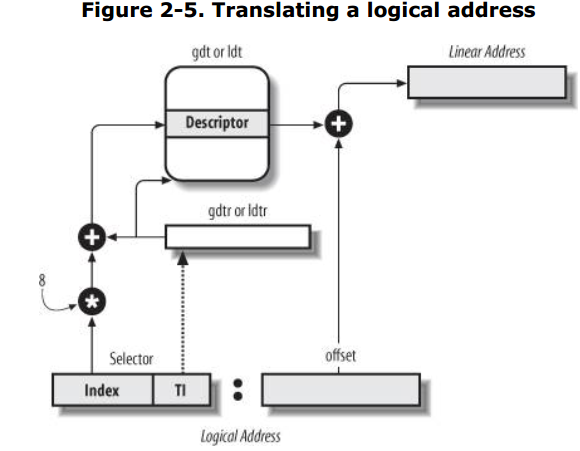
物理地址：用于内存芯片级内存单元寻址。它们与从微处理器的地址引脚发送到内存总线上的电信号相对应。物理地址由32位或36位无符号整数表示。



### 分段单元

段寄存器的唯一目的是存放段选择符。

cs代码段寄存器，含有一个两位的字段RPL，指明cpu的当前特权级（current privilege level CPL）。



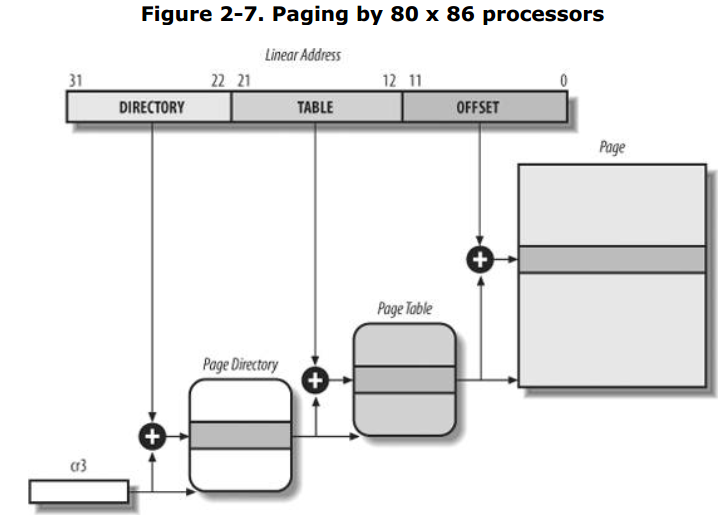
linux中段描述符中的base=0.线性地址和逻辑地址一致。

### 页表

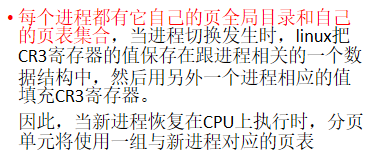
为了效率起见，线性地址被分成以固定长度为单位的组，称为页。页内连续的线性地址被映射到连续的物理地址中。

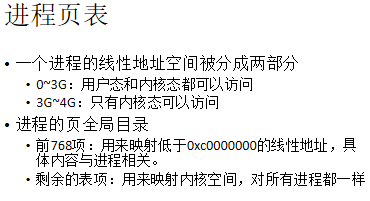
把线性地址映射到物理地址的数据结构叫做页表(page table)。页表存放在内存中，并在启用分页单元以前由内核对之进行初始化。

### 分页单元



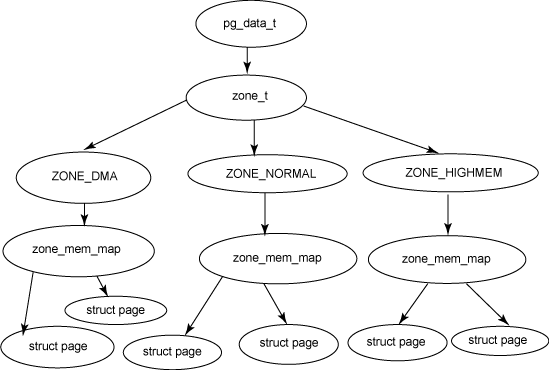
### 进程地址空间与进程页表





## **九、页框、管理区、节点**

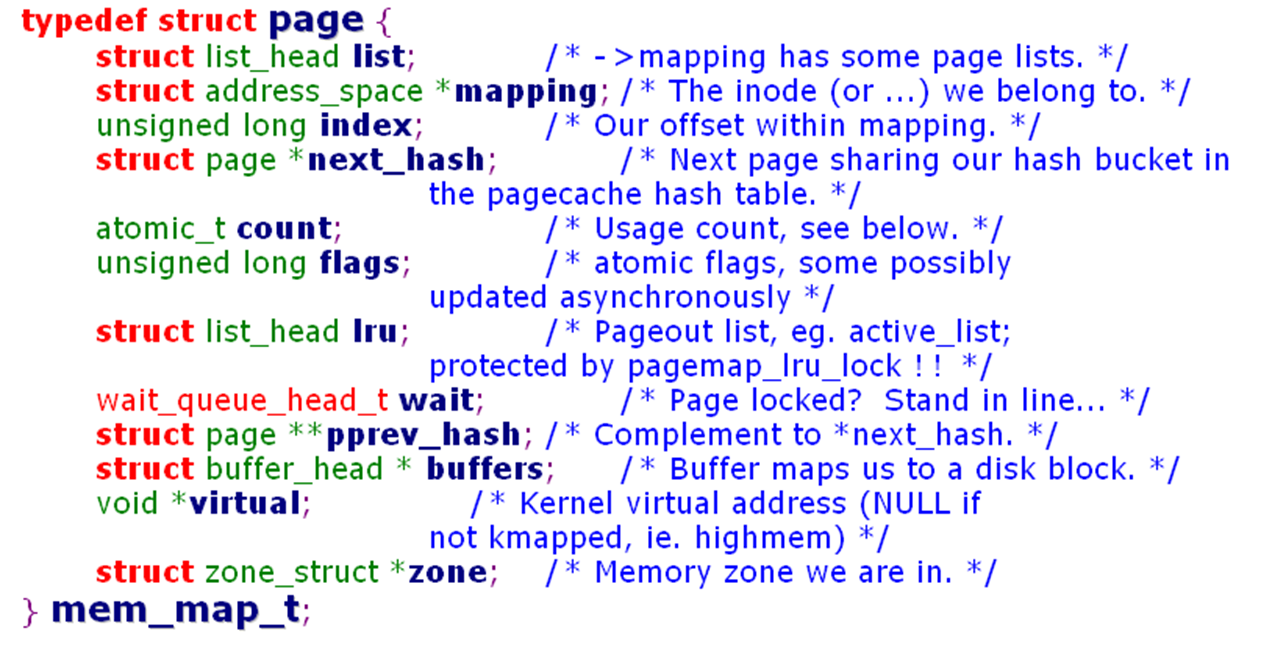
掌握页框、管理区、节点的主要数据结构 ，管理区对空闲页框的组织为选学内**容**

****

### page

内核使用页描述符来页框的状态信息，每个物理页框都用一个页描述符struct page表示。

所有物理页框的描述符，组织在mem\_map的数组中。

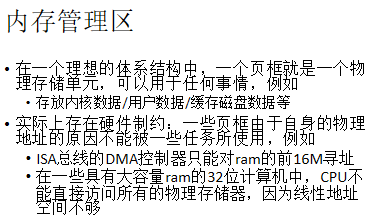


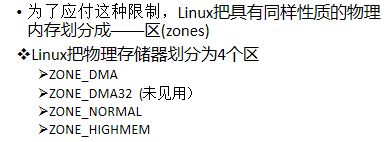
### 节点

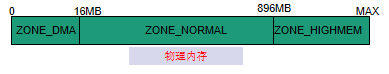


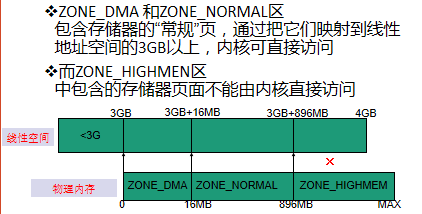


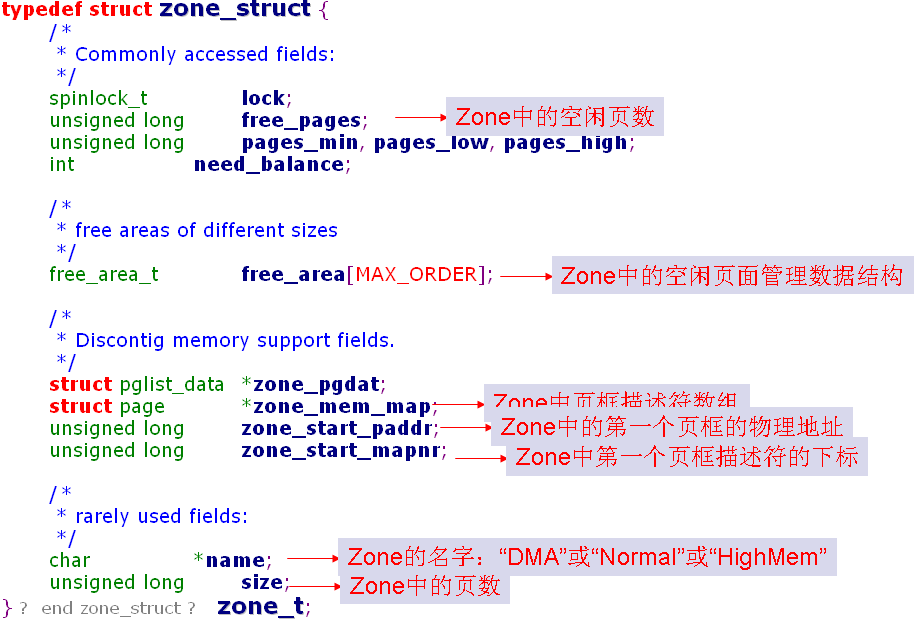
### 区











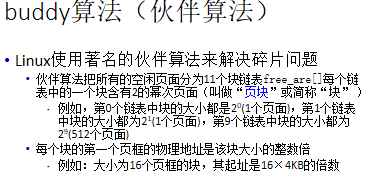
### 伙伴系统的基本原理

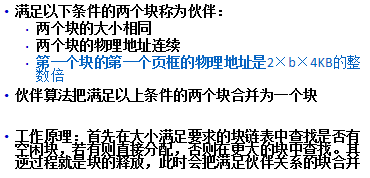
随着用户程序的执行和结束，系统需要不断为其分配和释放物理页面。但是，频繁地请求和释放不同大小的一组连续页面，必然导致在已分配的内存块中分撒着许多小块的空闲页面，即外碎片。由此带来的问题是，即使这些小块的空闲页面加起来足以满足所请求的页面，但是要分配一个大块的连续页面可能根本无法满足。

必须有一种方法来解决这一问题。内核应该为分配一组连续的页面而建立一种稳定、高效的分配策略。

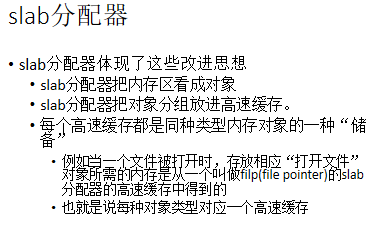
有两种办法可以避免这样的碎片，1.利用MMU把一组非连续的物理空闲页框映射到连续的线性地址空间；2.使用一种适当的技术来记录现存的空闲连续页框的情况，以尽量避免为满足对小块的请求而把大块的空闲块进行分割。

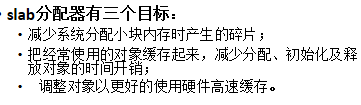
由于在某些情况下，必须使用连续的页框，如DMA，并且希望尽量少的修改内核页表。Linux内核首选第二种方法。

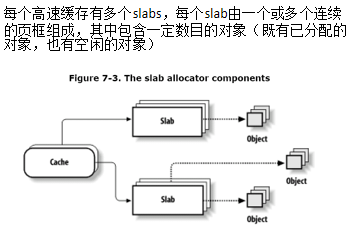


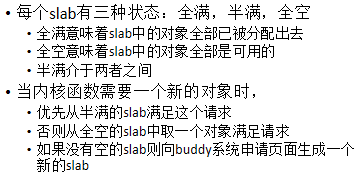


### slab分配器







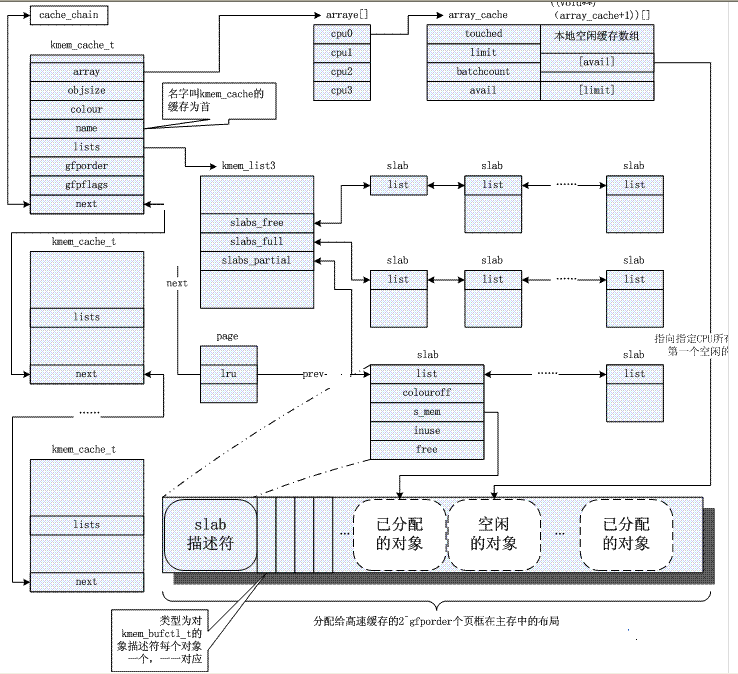


内核对内存分配的方式：伙伴系统、slab以及非连续区的作用

注意一个新创建的高速缓存没有包含任何slab，因此也没有空闲的对象。只有当以下两个条件都为真时，才给高速缓存分配slab：

* + - 已发出一个分配新对象的请求，即调用kmem\_cache\_alloc()请求对象；
    - 高速缓存不包含任何空闲对象。

明确slab分配器主要数据结构的关系

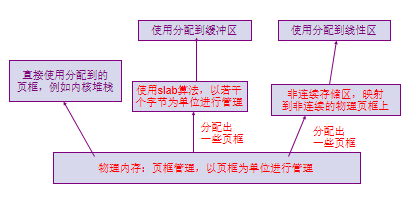


### 非连续区分配内存的内核的分配过程

vmalloc()

分配非连续叶框、修改内核使用的页表项。

### 页框管理、内存区管理、非连续内存区管理之间的关系



### 页框回收

Linux 系统在为用户态进程和内核分配动态内存的时候，所作的检查是马马虎虎的

对内核使用的许多磁盘高速缓存和内存高速缓存大小也同样不作限制。

页框回收算法（PFRA）：

1、在所有内存使用完之前，就必须执行页框回收算法

2、选择目标页，它获取页框，并且使之空闲

3、候选回收页：任何属于磁盘和内存高速缓存的页，以及属于进程用户态地址空间的页

4、首先释放‘无害’页：先释放没有被任何进程使用的磁盘与内存高速缓存中的页

5、将用户态进程的所有页定为可回收页。

6、同时取消引用一个共享页框的所有页表项的映射，就可以回收改共享页框

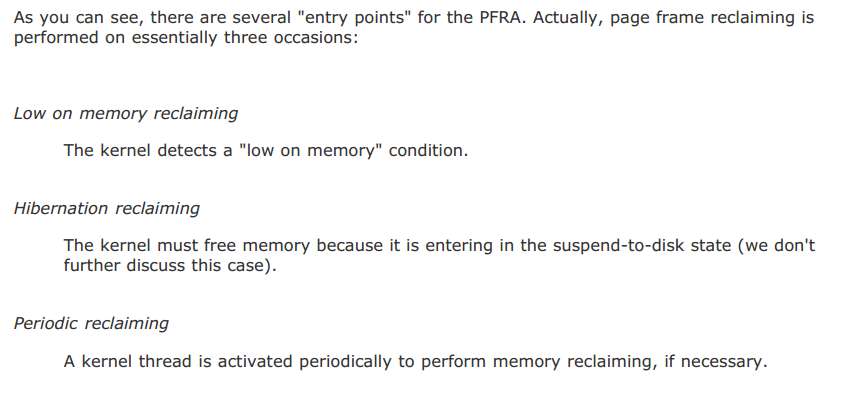
7、只回收未用页。

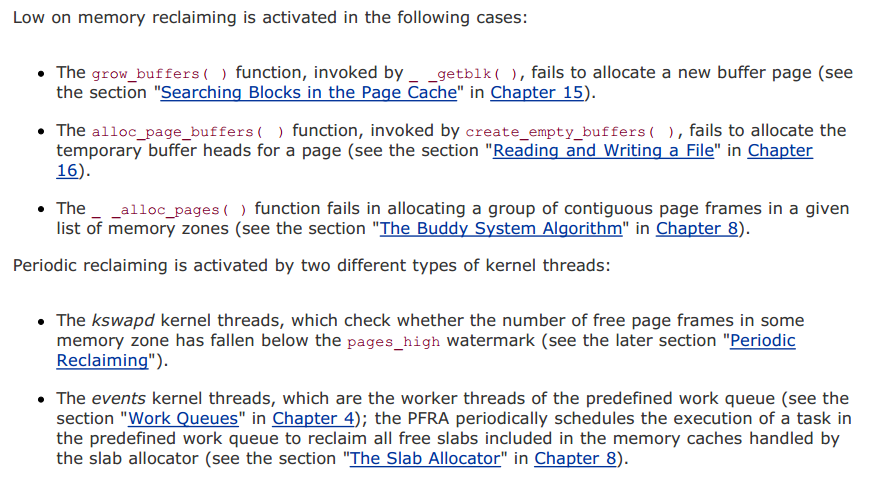
8、LRU算法回收任何进程最旧页。

反向映射：

1、内核能快速定位到同一页框到所有页表项，就叫反向映射。

2、采用面向对象的反向映射技术

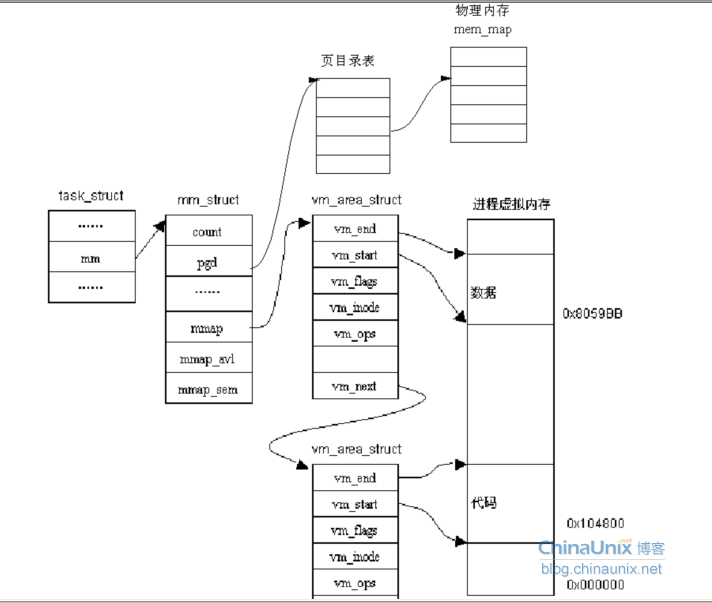




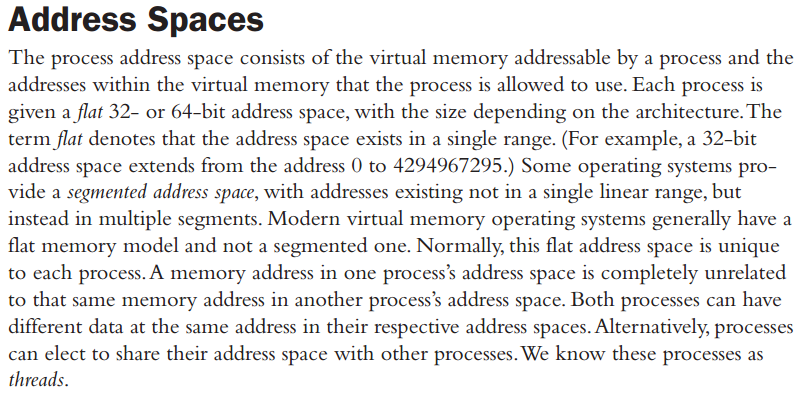
## 十、进程地址空间

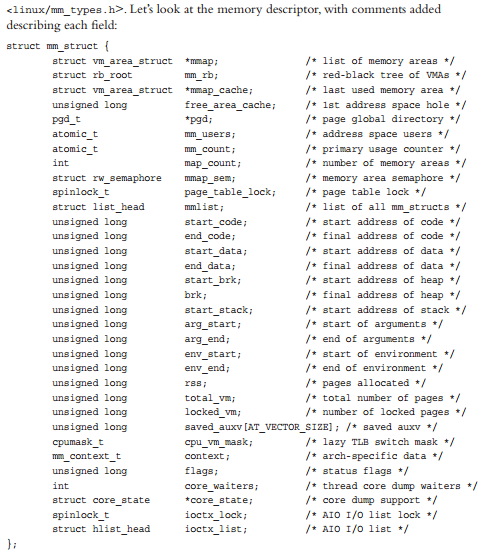
**进程地址空间、进程对线性区的组织、进程一般何时获得线性区？内核与普通进程获取内存时有何不同？何时为进程分配页框？明确请求调页和写时复制？进程线性区包含的主要内容？明确缺页异常的处理过程？**

**相关逻辑关系示意图**

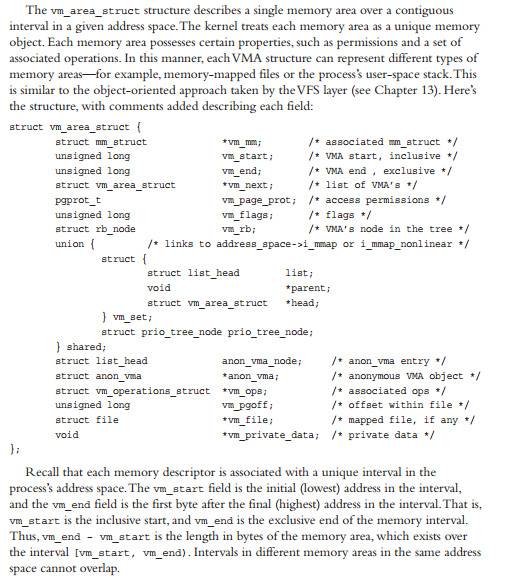


### mm\_struct





### vm\_area\_struct



### 进程对线性区的组织

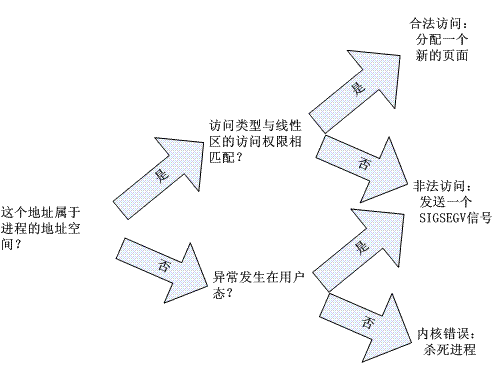
mmap指向的线性区链表用来遍历整个进程的地址空间。

红黑树mm\_rb用来定位一个给定的线性地址落在进程地址空间中的哪一个线性区中。

通过内存描述符中的两个域mmap和mm\_rb都可以访问线性区。事实上，它们都指向了同一个vm\_area\_struct结构，只是链接的方式不同。

### 缺页异常

内核只是通过mmap()等调用分配了一些线性地址空间给进程，并没有真正的把实际的物理页框分配给进程。当进程试图访问这些分配给它的地址空间时，比如一段线性地址空间映射的是二进制代码，则进程被调度执行的时候会跳转到这个地址上去执行。此时，并没有物理页框对应于这些线性地址，从而会引发一个缺页异常。



### 请求调页

是一种动态内存分配技术，它把页框的分配推迟到进程要访问的页不在RAM中为止，由此引起缺页异常。

进程开始运行的时候并不访问其地址空间中的全部地址；事实上有一部分地址也许永远不被进程使用。

相对于全局分配，请求调页增加了系统中的空闲页框的平均数，使系统有更大的吞吐量。

### 写时复制

子进程复制父进程的整个地址空间非常耗时：

为子进程的页表分配页框

为进程的页分配页框

初始化子进程的页表

把父进程的页复制到子进程相应的页中

这种创建地址空间的方法涉及许多内存访问，消耗许多CPU周期；并且这样做经常毫无意义。许多子进程通过装入一个新的程序开始它们的执行，这样就完全丢弃了所继承的空间。

写时复制：

父进程和子进程共享页框而不是复制页框。

共享的页框不能被修改。

父进程和子进程何时试图写一个共享的页框，就产生一个异常，这时内核就把这个页复制到一个新的页框中并标记为可写。

原来的共享页框仍然是写保护的：当其他进程试图写入时，内核检查写进程是否是这个页框的唯一属主，如果是就把这个页框分配给它。

这样一段时间后进程就会有自己的空间，并且按需分配。

### 创建进程地址空间

当创建一个新的进程时内核调用copy\_mm（）函数；该函数通过建立新进程的所有页表和内存描述符来创建进程的地址空间。

复制当前进程的局部描述符表并把它插入创建进程的进程描述符中。

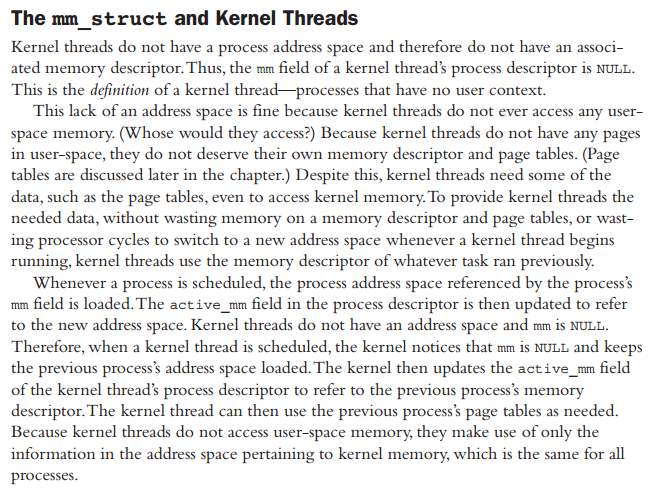
复制进程的线性区。

把新进程的内存描述符插入到内存描述符全局链表中。

扫描父进程的线性区链表，复制链表中的线性区描述符到子进程的线性区链表和红黑树中。

复制父进程页表，并把页表所对应的页框标记为对父子进程是只读的。

### mm\_struct与内核线程



copy\_mm函数把父进程的地址空间给子进程，如果带有CLONE\_VM标志，则子进程共享父进程的mm\_struct；如果该标志没有被设置，copy\_mm函数必须创建一个新的地址空间（申请并初始化新的mm\_struct结构，并赋值给子进程的mm字段）。

### 内核态和用户态分配内存的不同

内核中的函数以直接了当的方式获得动态内存

内核是操作系统中优先级最高的成分。

内核信任自己

采用前面介绍的页面级内存分配和小内存分配以及非连续线性区

给用户态进程分配内存时

请求被认为是不紧迫的

用户进程不可信任

因此，当用户态进程请求动态内存时，并没有立即获得实际的物理页框，而仅仅获得对一个新的线性地址区间的使用权

这个线性地址区间会成为进程地址空间的一部分，称作线性区(memory areas)

### 进程何时会获得新的线性区

(1) 当创建一个新的进程时，一个全新的地址空间被分配给了新的进程。

(2) 正在运行的进程装入新的程序（exec）时，旧的线性区被释放，新的线性区被分配给进程。

(3) 进程持续向用户态堆栈增加数据。

(4) 进程通过malloc扩展动态区。

(5) 进程创建一个IPC共享线性区与其他合作进程共享数据。

与创建、删除线性区相关的系统调用主要包括brk(), execve(), \_exit(), fork(), mmap(), mmap2(), mremap(), munmap(), shmat(), shmdt()等。