**Linux**内存管理

**摘要**：本章首先以应用程序开发者的角度审视Linux的进程内存管理，在此基础上逐步深入到内核中讨论系统物理内存管理和内核内存的使用方法。力求从外到内、水到渠成地引导网友分析Linux的内存管理与使用。在本章最后，我们给出一个内存映射的实例，帮助网友们理解内核内存管理与用户内存管理之间的关系，希望大家最终能驾驭Linux内存管理。

**前言**

内存管理一向是所有操作系统书籍不惜笔墨重点讨论的内容，无论市面上或是网上都充斥着大量涉及内存管理的教材和资料。因此，我们这里所要写的Linux内存管理采取避重就轻的策略，从理论层面就不去班门弄斧，贻笑大方了。我们最想做的和可能做到的是从开发者的角度谈谈对内存管理的理解，最终目的是把我们在内核开发中使用内存的经验和对Linux内存管理的认识与大家共享。

当然，这其中我们也会涉及到一些诸如段页等内存管理的基本理论，但我们的目的不是为了强调理论，而是为了指导理解开发中的实践，所以仅仅点到为止，不做深究。

遵循“理论来源于实践”的“教条”，我们先不必一下子就钻入内核里去看系统内存到底是如何管理，那样往往会让你陷入似懂非懂的窘境（我当年就犯了这个错误！）。所以最好的方式是先从外部（用户编程范畴）来观察进程如何使用内存，等到大家对内存的使用有了较直观的认识后，再深入到内核中去学习内存如何被管理等理论知识。最后再通过一个实例编程将所讲内容融会贯通。

**进程与内存**

**进程如何使用内存？**

毫无疑问，所有进程（执行的程序）都必须占用一定数量的内存，它或是用来存放从磁盘载入的程序代码，或是存放取自用户输入的数据等等。不过进程对这些内存的管理方式因内存用途不一而不尽相同，有些内存是事先静态分配和统一回收的，而有些却是按需要动态分配和回收的。

对任何一个普通进程来讲，它都会涉及到5种不同的数据段。稍有编程知识的朋友都能想到这几个数据段中包含有“程序代码段”、“程序数据段”、“程序堆栈段”等。不错，这几种数据段都在其中，但除了以上几种数据段之外，进程还另外包含两种数据段。下面我们来简单归纳一下进程对应的内存空间中所包含的5种不同的数据区。

***代码段***：代码段是用来存放可执行文件的操作指令，也就是说是它是可执行程序在内存中的镜像。代码段需要防止在运行时被非法修改，所以只准许读取操作，而不允许写入（修改）操作——它是不可写的。

***数据段***：数据段用来存放可执行文件中已初始化全局变量，换句话说就是存放程序静态分配[1]的变量和全局变量。

***BSS段*[2]**：BSS段包含了程序中未初始化的全局变量，在内存中 bss段全部置零。

***堆（*heap*）***：堆是用于存放进程运行中被动态分配的内存段，它的大小并不固定，可动态扩张或缩减。当进程调用malloc等函数分配内存时，新分配的内存就被动态添加到堆上（堆被扩张）；当利用free等函数释放内存时，被释放的内存从堆中被剔除（堆被缩减）

***栈***：栈是用户存放程序临时创建的局部变量，也就是说我们函数括弧“{}”中定义的变量（但不包括static声明的变量，static意味着在数据段中存放变量）。除此以外，在函数被调用时，其参数也会被压入发起调用的进程栈中，并且待到调用结束后，函数的返回值也会被存放回栈中。由于栈的先进先出特点，所以栈特别方便用来保存/恢复调用现场。从这个意义上讲，我们可以把堆栈看成一个寄存、交换临时数据的内存区。

**进程如何组织这些区域？**

上述几种内存区域中数据段、BSS和堆通常是被连续存储的——内存位置上是连续的，而代码段和栈往往会被独立存放。有趣的是，堆和栈两个区域关系很“暧昧”，他们一个向下“长”（i386体系结构中栈向下、堆向上），一个向上“长”，相对而生。但你不必担心他们会碰头，因为他们之间间隔很大（到底大到多少，你可以从下面的例子程序计算一下），绝少有机会能碰到一起。

下图简要描述了进程内存区域的分布：



“事实胜于雄辩”，我们用一个小例子（原形取自《User-Level Memory Management》）来展示上面所讲的各种内存区的差别与位置。

*#include<stdio.h*>

*#include<malloc.h*>

*#include<unistd.h*>

*int bss\_var*;

*int data\_var0=1;*

*int main(int* argc,char \*\*argv)

*{*

printf("below are addresses of types of process's mem\n");

printf("Text location:\n");

printf("\tAddress of main(Code Segment):%p\n",main);

printf("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n");

int stack\_var0=2;

printf("Stack Location:\n");

printf("\tInitial end of stack:%p\n",&stack\_var0);

int stack\_var1=3;

printf("\tnew end of stack:%p\n",&stack\_var1);

printf("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n");

printf("Data Location:\n");

printf("\tAddress of data\_var(Data Segment):%p\n",&data\_var0);

static int data\_var1=4;

printf("\tNew end of data\_var(Data Segment):%p\n",&data\_var1);

printf("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n");

printf("BSS Location:\n");

printf("\tAddress of bss\_var:%p\n",&bss\_var);

printf("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n");

char \*b = sbrk((ptrdiff\_t)0);

printf("Heap Location:\n");

printf("\tInitial end of heap:%p\n",b);

brk(b+4);

b=sbrk((ptrdiff\_t)0);

printf("\tNew end of heap:%p\n",b);

return 0;

}

它的结果如下

below are addresses of types of process's mem

Text location:

Address of main(Code Segment):0x8048388

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Stack Location:

Initial end of stack:0xbffffab4

new end of stack:0xbffffab0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Data Location:

Address of data\_var(Data Segment):0x8049758

New end of data\_var(Data Segment):0x804975c

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

BSS Location:

Address of bss\_var:0x8049864

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Heap Location:

Initial end of heap:0x8049868

New end of heap:0x804986c

利用size命令也可以看到程序的各段大小，比如执行size example会得到

text data bss dec hex filename

1654 280 8 1942 796 example

但这些数据是程序编译的静态统计，而上面显示的是进程运行时的动态值，但两者是对应的。

通过前面的例子，我们对进程使用的逻辑内存分布已先睹为快。这部分我们就继续进入操作系统内核看看，进程对内存具体是如何进行分配和管理的。

从用户向内核看，所使用的内存表象形式会依次经历“逻辑地址”——“线性地址”——“物理地址”几种形式（关于几种地址的解释在前面已经讲述了）。逻辑地址经段机制转化成线性地址；线性地址又经过页机制转化为物理地址。（但是我们要知道Linux系统虽然保留了段机制，但是将所有程序的段地址都定死为0-4G，所以虽然逻辑地址和线性地址是两种不同的地址空间，但在Linux中逻辑地址就等于线性地址，它们的值是一样的）。沿着这条线索，我们所研究的主要问题也就集中在下面几个问题。

1. 进程空间地址如何管理？

2. 进程地址如何映射到物理内存？

3. 物理内存如何被管理？

以及由上述问题引发的一些子问题。如系统虚拟地址分布；内存分配接口；连续内存分配与非连续内存分配等。

**进程内存空间**

Linux操作系统采用虚拟内存管理技术，使得每个进程都有各自互不干涉的进程地址空间。该空间是块大小为4G的线性虚拟空间，用户所看到和接触到的都是该虚拟地址，无法看到实际的物理内存地址。利用这种虚拟地址不但能起到保护操作系统的效果（用户不能直接访问物理内存），而且更重要的是，用户程序可使用比实际物理内存更大的地址空间（具体的原因请看硬件基础部分）。

在讨论进程空间细节前，这里先要澄清下面几个问题：

l 第一、4G的进程地址空间被人为的分为两个部分——用户空间与内核空间。用户空间从0到3G（0xC0000000），内核空间占据3G到4G。用户进程通常情况下只能访问用户空间的虚拟地址，不能访问内核空间虚拟地址。只有用户进程进行系统调用（代表用户进程在内核态执行）等时刻可以访问到内核空间。

l 第二、用户空间对应进程，所以每当进程切换，用户空间就会跟着变化；而内核空间是由内核负责映射，它并不会跟着进程改变，是固定的。内核空间地址有自己对应的页表（init\_mm.pgd），用户进程各自有不同的页表。

l 第三、每个进程的用户空间都是完全独立、互不相干的。不信的话，你可以把上面的程序同时运行10次（当然为了同时运行，让它们在返回前一同睡眠100秒吧），你会看到10个进程占用的线性地址一模一样。

**进程内存管理**

进程内存管理的对象是进程线性地址空间上的内存镜像，这些内存镜像其实就是进程使用的虚拟内存区域（memory region）。进程虚拟空间是个32或64位的“平坦”（独立的连续区间）地址空间（空间的具体大小取决于体系结构）。要统一管理这么大的平坦空间可绝非易事，为了方便管理，虚拟空间被划分为许多大小可变的(但必须是4096的倍数)内存区域，这些区域在进程线性地址中像停车位一样有序排列。这些区域的划分原则是“将访问属性一致的地址空间存放在一起”，所谓访问属性在这里无非指的是“可读、可写、可执行等”。

如果你要查看某个进程占用的内存区域，可以使用命令cat /proc/<pid>/maps获得（pid是进程号，你可以运行上面我们给出的例子——./example &;pid便会打印到屏幕），你可以发现很多类似于下面的数字信息。

由于程序example使用了动态库，所以除了example本身使用的的内存区域外，还会包含那些动态库使用的内存区域（区域顺序是：代码段、数据段、bss段）。

我们下面只抽出和example有关的信息，除了前两行代表的代码段和数据段外，最后一行是进程使用的栈空间。

-------------------------------------------------------------------------------

08048000 - 08049000 r-xp 00000000 03:03 439029 /home/mm/src/example

08049000 - 0804a000 rw-p 00000000 03:03 439029 /home/mm/src/example

……………

bfffe000 - c0000000 rwxp ffff000 00:00 0

----------------------------------------------------------------------------------------------------------------------

每行数据格式如下：

（内存区域）开始－结束 访问权限 偏移 主设备号：次设备号 i节点 文件。

注意，你一定会发现进程空间只包含三个内存区域，似乎没有上面所提到的堆、bss等，其实并非如此，程序内存段和进程地址空间中的内存区域是种模糊对应，也就是说，堆、bss、数据段（初始化过的）都在进程空间中由数据段内存区域表示。

在Linux内核中对应进程内存区域的数据结构是: vm\_area\_struct, 内核将每个内存区域作为一个单独的内存对象管理，相应的操作也都一致。采用面向对象方法使VMA结构体可以代表多种类型的内存区域－－比如内存映射文件或进程的用户空间栈等，对这些区域的操作也都不尽相同。

vm\_area\_strcut结构比较复杂，关于它的详细结构请参阅相关资料。我们这里只对它的组织方法做一点补充说明。vm\_area\_struct是描述进程地址空间的基本管理单元，对于一个进程来说往往需要多个内存区域来描述它的虚拟空间，如何关联这些不同的内存区域呢？大家可能都会想到使用链表，的确vm\_area\_struct结构确实是以链表形式链接，不过为了方便查找，内核又以红黑树（以前的内核使用平衡树）的形式组织内存区域，以便降低搜索耗时。并存的两种组织形式，并非冗余：链表用于需要遍历全部节点的时候用，而红黑树适用于在地址空间中定位特定内存区域的时候。内核为了内存区域上的各种不同操作都能获得高性能，所以同时使用了这两种数据结构。

下图反映了进程地址空间的管理模型：



进程的地址空间对应的描述结构是“内存描述符结构”,它表示进程的全部地址空间，——包含了和进程地址空间有关的全部信息，其中当然包含进程的内存区域。

**进程内存的分配与回收**

创建进程fork()、程序载入exec()、映射文件mmap()、动态内存分配malloc()/brk()等进程相关操作都需要分配内存给进程。不过这时进程申请和获得的还不是实际内存，而是虚拟内存，准确的说是“内存区域”。进程对内存区域的分配最终都会归结到do\_mmap（）函数上来（brk调用被单独以系统调用实现，不用do\_mmap()），

内核使用do\_mmap()函数创建一个新的线性地址区间。但是说该函数创建了一个新VMA并不非常准确，因为如果创建的地址区间和一个已经存在的地址区间相邻，并且它们具有相同的访问权限的话，那么两个区间将合并为一个。如果不能合并，那么就确实需要创建一个新的VMA了。但无论哪种情况， do\_mmap()函数都会将一个地址区间加入到进程的地址空间中－－无论是扩展已存在的内存区域还是创建一个新的区域。

同样，释放一个内存区域应使用函数do\_ummap()，它会销毁对应的内存区域。

**如何由虚变实！**

从上面已经看到进程所能直接操作的地址都为虚拟地址。当进程需要内存时，从内核获得的仅仅是虚拟的内存区域，而不是实际的物理地址，进程并没有获得物理内存（物理页面——页的概念请大家参考硬件基础一章），获得的仅仅是对一个新的线性地址区间的使用权。实际的物理内存只有当进程真的去访问新获取的虚拟地址时，才会由“请求页机制”产生“缺页”异常，从而进入分配实际页面的例程。

该异常是虚拟内存机制赖以存在的基本保证——它会告诉内核去真正为进程分配物理页，并建立对应的页表，这之后虚拟地址才实实在在地映射到了系统的物理内存上。（当然，如果页被换出到磁盘，也会产生缺页异常，不过这时不用再建立页表了）

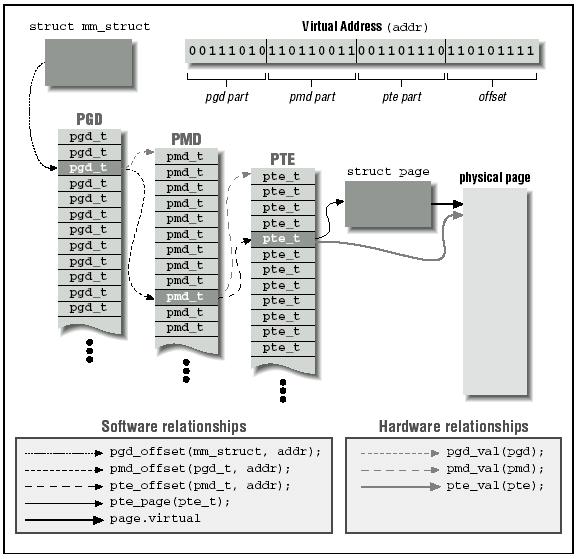
这种请求页机制把页面的分配推迟到不能再推迟为止，并不急于把所有的事情都一次做完（这种思想有点像设计模式中的代理模式（proxy））。之所以能这么做是利用了内存访问的“局部性原理”，请求页带来的好处是节约了空闲内存，提高了系统的吞吐率。要想更清楚地了解请求页机制，可以看看《深入理解linux内核》一书。

这里我们需要说明在内存区域结构上的nopage操作。当访问的进程虚拟内存并未真正分配页面时，该操作便被调用来分配实际的物理页，并为该页建立页表项。在最后的例子中我们会演示如何使用该方法。

**系统物理内存管理**

虽然应用程序操作的对象是映射到物理内存之上的虚拟内存，但是处理器直接操作的却是物理内存。所以当应用程序访问一个虚拟地址时，首先必须将虚拟地址转化成物理地址，然后处理器才能解析地址访问请求。地址的转换工作需要通过查询页表才能完成，概括地讲，地址转换需要将虚拟地址分段，使每段虚地址都作为一个索引指向页表，而页表项则指向下一级别的页表或者指向最终的物理页面。

每个进程都有自己的页表。进程描述符的pgd域指向的就是进程的页全局目录。下面我们借用《linux设备驱动程序》中的一幅图大致看看进程地址空间到物理页之间的转换关系。



上面的过程说起来简单，做起来难呀。因为在虚拟地址映射到页之前必须先分配物理页——也就是说必须先从内核中获取空闲页，并建立页表。下面我们介绍一下内核管理物理内存的机制。

**物理内存管理（页管理）**

Linux内核管理物理内存是通过分页机制实现的，它将整个内存划分成无数个4k（在i386体系结构中）大小的页，从而分配和回收内存的基本单位便是**内存页了**。利用分页管理有助于灵活分配内存地址，因为分配时不必要求必须有大块的连续内存[3]，系统可以东一页、西一页的凑出所需要的内存供进程使用。虽然如此，但是实际上系统使用内存时还是倾向于分配连续的内存块，因为分配连续内存时，页表不需要更改，因此能降低TLB的刷新率（频繁刷新会在很大程度上降低访问速度）。

鉴于上述需求，内核分配物理页面时为了尽量**减少不连续情况，采用了“伙伴”关系来管理空闲页面**。伙伴关系分配算法大家应该不陌生——几乎所有操作系统方面的书都会提到,我们不去详细说它了，如果不明白可以参看有关资料。这里只需要大家明白Linux中空闲页面的组织和管理利用了伙伴关系，因此空闲页面分配时也需要遵循伙伴关系，最小单位只能是2的幂倍页面大小。**内核中分配空闲页面的基本函数是get\_free\_page/get\_free\_pages**，它们或是分配单页或是分配指定的页面（2、4、8…512页）。

注意：get\_free\_page是在内核中分配内存，不同于malloc在用户空间中分配，malloc利用堆动态分配，实际上是调用brk()系统调用，该调用的作用是扩大或缩小进程堆空间（它会修改进程的brk域）。如果现有的内存区域不够容纳堆空间，则会以页面大小的倍数为单位，扩张或收缩对应的内存区域，但brk值并非以页面大小为倍数修改，而是按实际请求修改。因此Malloc在用户空间分配内存可以以字节为单位分配,但内核在内部仍然会是以页为单位分配的。

另外,需要提及的是，物理页在系统中由页结构struct page描述，系统中所有的页面都存储在数组mem\_map[]中，可以通过该数组找到系统中的每一页（空闲或非空闲）。而其中的空闲页面则可由上述提到的以伙伴关系组织的空闲页链表（free\_area[MAX\_ORDER]）来索引。



**内核内存使用**

**Slab**

所谓尺有所长，寸有所短。以页为最小单位分配内存对于内核管理系统中的物理内存来说的确比较方便，但**内核自身最常使**用的内存却往往是很小（远远小于一页）的内存块——比如存放文件描述符、进程描述符、虚拟内存区域描述符等行为所需的内存都不足一页。这些用来存放描述符的内存相比页面而言，就好比是面包屑与面包。一个整页中可以聚集多个这些小块内存；而且这些小块内存块也和面包屑一样频繁地生成/销毁。

为了满足内核对这种小内存块的需要，Linux系统采用了一种被称为slab分配器的技术。Slab分配器的实现相当复杂，但原理不难，其核心思想就是“存储池[4]”的运用。内存片段（小块内存）被看作对象，当被使用完后，并不直接释放而是被缓存到“存储池”里，留做下次使用，这无疑避免了频繁创建与销毁对象所带来的额外负载。

Slab技术不但避免了内存内部分片（下文将解释）带来的不便（引入Slab分配器的主要目的是为了减少对伙伴系统分配算法的调用次数——频繁分配和回收必然会导致内存碎片——难以找到大块连续的可用内存），而且可以很好地利用硬件缓存提高访问速度。

Slab并非是脱离伙伴关系而独立存在的一种内存分配方式，slab仍然是建立在页面基础之上，换句话说，Slab将页面（来自于伙伴关系管理的空闲页面链表）撕碎成众多小内存块以供分配，slab中的对象分配和销毁使用kmem\_cache\_alloc与kmem\_cache\_free。

**Kmalloc**

Slab分配器不仅仅只用来存放内核专用的结构体，它还被用来处理内核对小块内存的请求。当然鉴于Slab分配器的特点，一般来说**内核程序中对小于一页的小块内存的请求才通过Slab分配器提供的接口Kmalloc来完成**（虽然它可分配32 到131072字节的内存）。从内核内存分配的角度来讲，kmalloc可被看成是get\_free\_page（s）的一个有效补充，内存分配粒度更灵活了。

有兴趣的话，可以到/proc/slabinfo中找到内核执行现场使用的各种slab信息统计，其中你会看到系统中所有slab的使用信息。从信息中可以看到系统中除了专用结构体使用的slab外，还存在大量为Kmalloc而准备的Slab（其中有些为dma准备的）。

**内核非连续内存分配（Vmalloc）**

伙伴关系也好、slab技术也好，从内存管理理论角度而言目的基本是一致的，它们都是为了防止“分片”，不过分片又分为**外部分片和内部分片之说**，所谓内部分片是说系统为了满足一小段内存区（连续）的需要，不得不分配了一大区域连续内存给它，从而造成了空间浪费；外部分片是指系统虽有足够的内存，但却是分散的碎片，无法满足对大块“连续内存”的需求。无论何种分片都是系统有效利用内存的障碍。slab分配器使得一个页面内包含的众多小块内存可独立被分配使用，避免了内部分片，节约了空闲内存。伙伴关系把内存块按大小分组管理，一定程度上减轻了外部分片的危害，因为页框分配不在盲目，而是按照大小依次有序进行，**不过伙伴关系只是减轻了外部分片，但并未彻底消除。**你自己比划一下多次分配页面后，空闲内存的剩余情况吧。

所以避免外部分片的最终思路还是落到了如何利用不连续的内存块组合成“看起来很大的内存块”——这里的情况很类似于用户空间分配虚拟内存，内存逻辑上连续，其实映射到并不一定连续的物理内存上。**Linux内核借用了这个技术，允许内核程序在内核地址空间中分配虚拟地址，同样也利用页表（内核页表）将虚拟地址映射到分散的内存页上**。以此完美地解决了内核内存使用中的外部分片问题。内核提供vmalloc函数分配内核虚拟内存，该函数不同于kmalloc，它可以分配较Kmalloc大得多的内存空间（可远大于128K，但必须是页大小的倍数），但相比Kmalloc来说,Vmalloc需要对内核虚拟地址进行重映射，必须更新内核页表，因此分配效率上要低一些（用空间换时间）0

与用户进程相似,内核也有一个名为init\_mm的mm\_strcut结构来描述内核地址空间，其中页表项pdg=swapper\_pg\_dir包含了系统内核空间（3G-4G）的映射关系。因此vmalloc分配内核虚拟地址必须更新内核页表，而kmalloc或get\_free\_page由于分配的连续内存，所以不需要更新内核页表。



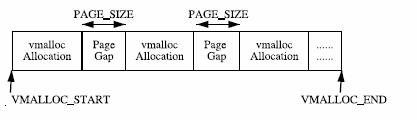
vmalloc分配的内核虚拟内存与kmalloc/get\_free\_page分配的内核虚拟内存位于不同的区间，不会重叠。因为内核虚拟空间被分区管理，各司其职。进程空间地址分布从０到３G(其实是到PAGE\_OFFSET, 在0x86中它等于0xC0000000)，从3G到vmalloc\_start这段地址是物理内存映射区域（该区域中包含了内核镜像、物理页面表mem\_map等等）比如我使用的系统内存是64M(可以用free看到)，那么(3G——3G+64M)这片内存就应该映射到物理内存，而vmalloc\_start位置应在3G+64M附近（说"附近"因为是在物理内存映射区与vmalloc\_start期间还会存在一个8M大小的gap来防止跃界）,vmalloc\_end的位置接近4G(说"接近"是因为最后位置系统会保留一片128k大小的区域用于专用页面映射，还有可能会有高端内存映射区，这些都是细节，这里我们不做纠缠)。



上图是内存分布的模糊轮廓

　 由get\_free\_page或Kmalloc函数所分配的连续内存都陷于物理映射区域，所以它们返回的内核虚拟地址和实际物理地址仅仅是相差一个偏移量（PAGE\_OFFSET），你可以很方便的将其转化为物理内存地址，同时内核也提供了virt\_to\_phys（）函数将内核虚拟空间中的物理映射区地址转化为物理地址。要知道，物理内存映射区中的地址与内核页表是有序对应的，系统中的每个物理页面都可以找到它对应的内核虚拟地址（在物理内存映射区中的）。

而vmalloc分配的地址则限于vmalloc\_start与vmalloc\_end之间。每一块vmalloc分配的内核虚拟内存都对应一个vm\_struct结构体（可别和vm\_area\_struct搞混，那可是进程虚拟内存区域的结构），不同的内核虚拟地址被4k大小的空闲区间隔，以防止越界——见下图）。与进程虚拟地址的特性一样，这些虚拟地址与物理内存没有简单的位移关系，必须通过内核页表才可转换为物理地址或物理页。它们有可能尚未被映射，在发生缺页时才真正分配物理页面。



这里给出一个小程序帮助大家认清上面几种分配函数所对应的区域。

#include<linux/module.h>

#include<linux/slab.h>

#include<linux/vmalloc.h>

unsigned char \*pagemem;

unsigned char \*kmallocmem;

unsigned char \*vmallocmem;

int init\_module(void)

{

pagemem = get\_free\_page(0);

printk("<1>pagemem=%s",pagemem);

kmallocmem = kmalloc(100,0);

printk("<1>kmallocmem=%s",kmallocmem);

vmallocmem = vmalloc(1000000);

printk("<1>vmallocmem=%s",vmallocmem);

}

void cleanup\_module(void)

{

free\_page(pagemem);

kfree(kmallocmem);

vfree(vmallocmem);

}

**实例**

内存映射(mmap)是Linux操作系统的一个很大特色，它可以将系统内存映射到一个文件（设备）上，以便可以通过访问文件内容来达到访问内存的目的。这样做的最大好处是提高了内存访问速度，并且可以利用文件系统的接口编程（设备在Linux中作为特殊文件处理）访问内存，降低了开发难度。许多设备驱动程序便是利用内存映射功能将用户空间的一段地址关联到设备内存上，无论何时，只要内存在分配的地址范围内进行读写，实际上就是对设备内存的访问。同时对设备文件的访问也等同于对内存区域的访问，也就是说，通过文件操作接口可以访问内存。Linux中的X服务器就是一个利用内存映射达到直接高速访问视频卡内存的例子。

熟悉文件操作的朋友一定会知道file\_operations结构中有mmap方法，在用户执行mmap系统调用时，便会调用该方法来通过文件访问内存——不过在调用文件系统mmap方法前，内核还需要处理分配内存区域（vma\_struct）、建立页表等工作。对于具体映射细节不作介绍了，需要强调的是,建立页表可以采用remap\_page\_range方法一次建立起所有映射区的页表，或利用vma\_struct的nopage方法在缺页时现场一页一页的建立页表。第一种方法相比第二种方法简单方便、速度快， 但是灵活性不高。一次调用所有页表便定型了，不适用于那些需要现场建立页表的场合——比如映射区需要扩展或下面我们例子中的情况。

我们这里的实例希望利用内存映射，将系统内核中的一部分虚拟内存映射到用户空间，以供应用程序读取——你可利用它进行内核空间到用户空间的大规模信息传输。因此我们将试图写一个虚拟字符设备驱动程序，通过它将系统**内核空间映射到用户空间**——将内核虚拟内存映射到用户虚拟地址。从上一节已经看到Linux内核空间中包含两种虚拟地址：一种是物理和逻辑都连续的物理内存映射虚拟地址；另一种是逻辑连续但非物理连续的vmalloc分配的内存虚拟地址。我们的例子程序将演示把vmalloc分配的内核虚拟地址映射到用户地址空间的全过程。

程序里主要应解决两个问题：

第一是如何将vmalloc分配的内核虚拟内存正确地转化成物理地址？

因为内存映射先要获得被映射的物理地址，然后才能将其映射到要求的用户虚拟地址上。我们已经看到内核物理内存映射区域中的地址可以被内核函数virt\_to\_phys转换成实际的物理内存地址，但对于vmalloc分配的内核虚拟地址无法直接转化成物理地址，所以我们必须对这部分虚拟内存格外“照顾”——先将其转化成**内核物理内存映射区域中的地址，**然后在用virt\_to\_phys变为物理地址。

转化工作需要进行如下步骤：

a) 找到vmalloc虚拟内存对应的页表，并寻找到对应的页表项。

b) 获取页表项对应的页面指针

c) 通过页面得到对应的内核物理内存映射区域地址**。**

如下图所示：



第二是当访问vmalloc分配区时，如果发现虚拟内存尚未被映射到物理页，则需要处理“缺页异常”。因此需要我们实现内存区域中的nopaga操作，以能返回被映射的物理页面指针，在我们的实例中就是返回上面过程中的内核物理内存映射区域中的地址**。**由于vmalloc分配的虚拟地址与物理地址的对应关系并非分配时就可确定，必须在缺页现场建立页表，因此这里不能使用remap\_page\_range方法，只能用vma的nopage方法一页一页的建立。

**程序组成**

map\_driver.c，它是以模块形式加载的虚拟字符驱动程序。该驱动负责将一定长的内核虚拟地址(vmalloc分配的)映射到设备文件上。其中主要的函数有——vaddress\_to\_kaddress（）负责对vmalloc分配的地址进行页表解析,以找到对应的内核物理映射地址（kmalloc分配的地址）；map\_nopage()负责在进程访问一个当前并不存在的VMA页时，寻找该地址对应的物理页，并返回该页的指针。

test.c 它利用上述驱动模块对应的设备文件在用户空间读取读取内核内存。结果可以看到内核虚拟地址的内容（ok!），被显示在了屏幕上。

**执行步骤**

编译map\_driver.c为map\_driver.o模块,具体参数见Makefile

加载模块 ：insmod map\_driver.o

生成对应的设备文件

1 在/proc/devices下找到map\_driver对应的设备命和设备号：grep mapdrv /proc/devices

2 建立设备文件mknod mapfile c 254 0 （在我的系统里设备号为254）

利用maptest读取mapfile文件，将取自内核的信息打印到屏幕上。

全部程序下载 mmap.tar （感谢Martin Frey，该程序的主体出自他的灵感）

[1] 静态分配内存就是编译器在编译程序的时候根据源程序来分配内存. 动态分配内存就是在程序编译之后, 运行时调用运行时刻库函数来分配内存的. 静态分配由于是在程序运行之前,所以速度快, 效率高, 但是局限性大. 动态分配在程序运行时执行, 所以速度慢, 但灵活性高.

[2]术语"BSS"已经有些年头了，它是*block started by symbol*的缩写。因为未初始化的变量没有对应的值,所以并不需要存储在可执行对象中。但是因为C标准强制规定未初始化的全局变量要被赋予特殊的默认值(基本上是0值)，所以内核要从可执行代码装入变量(未赋值的)到内存中，然后将零页映射到该片内存上，于是这些未初始化变量就被赋予了0值。这样做避免了在目标文件中进行显式地初始化，减少空间浪费（来自《Linux内核开发》）

[3] 还有些情况必须要求内存连续，比如DMA传输中使用的内存，由于不涉及页机制所以必须连续分配。

[4] 这种存储池的思想在计算机科学里广泛应用，比如数据库连接池、内存访问池等等。

把linux内存管理分为下面四个层面

（一）硬件辅助的虚实地址转换

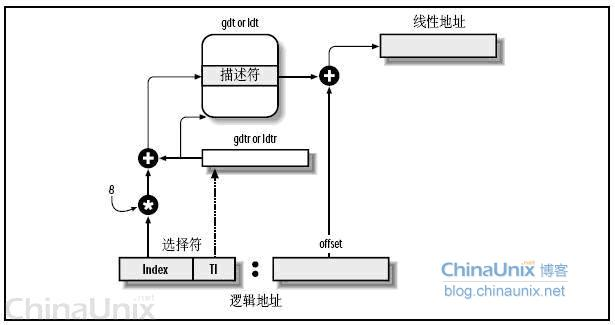
（二）内核管理的内存相关

（三）单个进程的内存管理

（四）malloc软件

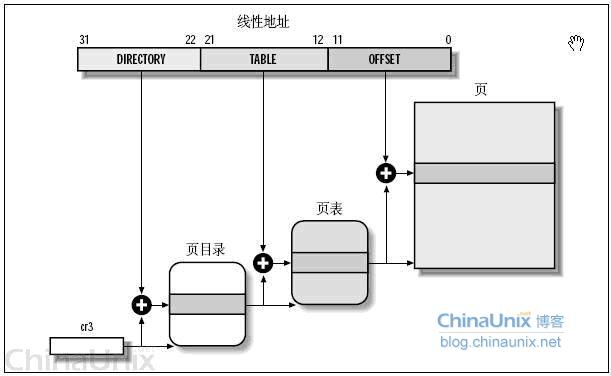
（一）           处理器硬件辅助的虚实地址转换（以x86为例）

在x86中虚实地址转换分为段式转换和页转换。段转换过程是由逻辑地址（或称为虚拟地址）转换为线性地址；页转换过程则是将线性地址转换为物理地址。段转换示意图如下



X86支持两种段，gdt和ldt（全局描述段表和局部描述符段表），在linux中只使用了4个全局描述符表，内核空间和用户空间分别两个gdt，分别对应各自的代码段和数据段。也可以认为在linux中变相地disable了x86的段式转换功能。

页转换示意图如下



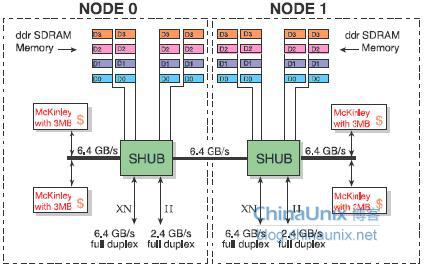
在linux中x86 的cr3寄存器（页表基地址寄存器）保存在进程的上下文中，在进程切换时会保存或回复该寄存器的内容，这样每个进程都有自己的转换页表，从而保证了每个进程有自己的虚拟空间。

（二）           内核管理的内存相关

从几个概念展开内存管理：node、zone、buddy、slab

1、Node

**SGI Altix3000系统的两个结点**

****

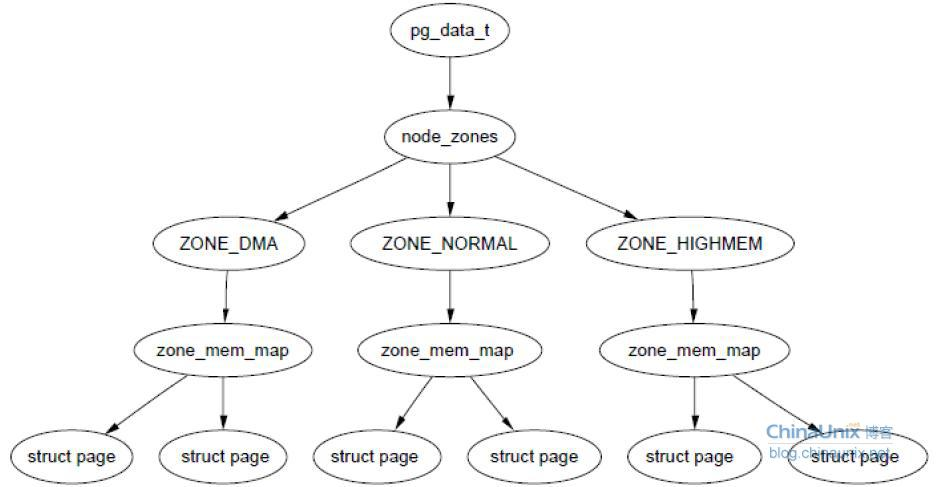
如上图，NUMA系统的结点通常是由一组CPU（如，SGI Altix 3000是2个Itanium2 CPU）和本地内存组成。由于每个结点都有自己的本地内存，因此全系统的内存在物理上是分布的，每个结点访问本地内存和访问其它结点的远地内存的延迟是不同的，为了优化对NUMA 系统的支持，引进了Node 来将NUMA 物理内存进行划分为不同的Node。而操作系统也必须能感知硬件的拓扑结构，优化系统的访存。

但是Intel x86 系统不是NUMA 系统。为了保持代码的一致性，在x86 平台上，Linux 将所有物理内存都划分到同一个Node。事实上，对于非NUMA 体系结构，也是如此处理的。

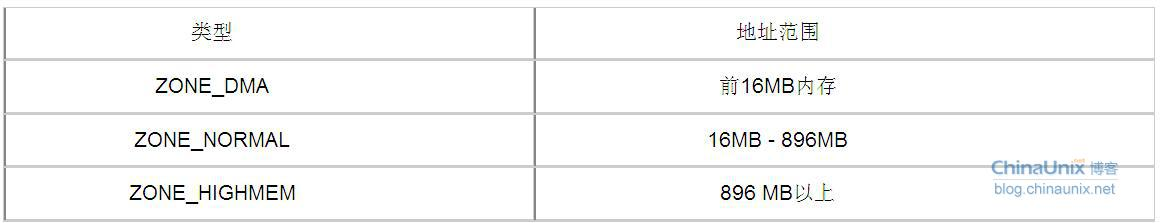
Linux系统用定义了数组pg\_data\_t node\_data[MAX\_NUMNODES] 来管理各个node。

2、Zone

**Linux中**Node**、**Zone**和页的关系**

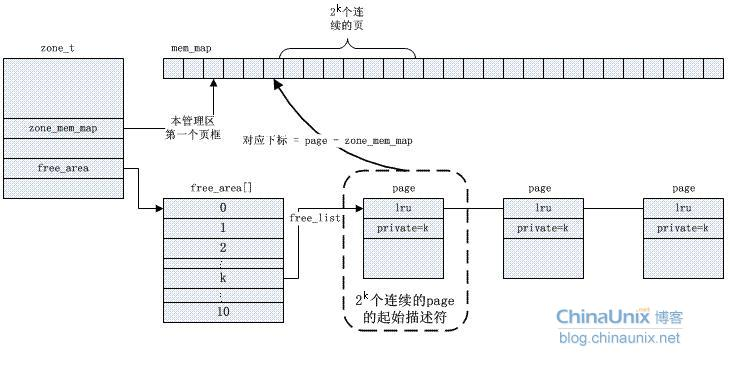
****

每个结点的内存被分为多个块，称为zones，它表示内存中一段区域。一个zone用struct zone结构描述，zone的类型主要有ZONE\_DMA、ZONE\_NORMAL和ZONE\_HIGHMEM。ZONE\_DMA位于低端的内存空间，用于某些旧的ISA设备。ZONE\_NORMAL的内存直接映射到Linux内核线性地址空间的高端部分，ZONE\_HIGHMEM位于物理地址高于896MB的区域。例如，在X86中，zone的物理地址如下：



内核空间只有1GB线性地址，如果使用大于1GB的物理内存就没法直接映射到内核线性空间了。当系统中的内存大于896MB时，把内核线性空间分为两部分，内核中低于896MB线性地址空间直接映射到低896MB的物理地址空间；高于896MB的128MB内核线性空间用于动态映射ZONE\_HIGHMEM内存区域（即物理地址高于896MB的物理空间）。

3、Buddy

如上

图所示，每个zone区域都采用伙伴系统（buddy system）来管理空闲内存页面。把所有的空闲页框分组为11个块链表，每个块链表分别包含大小为1，2，4，8，16，32，64，128，256，512和1024个连续的页框。链表编号分别为0，1，2，3，… k… 10。

从buddy system中申请页面过程：

1、根据申请存储区域大小查找对应的编号为K的块链表。

2、如果编号K的链表为空，则向编号为k+1的链表申请一个存储区域。如果编号为k+1链表不为空，系统从编号为k+1的链表上拆下一个区域，并将拆下的区域分为两个2^k的区域，一个返还给申请者，另一个则挂到编号为k的链表。

3、如果编号为k+1的链表也为空，编号为k+2的链表不为空。则从k+2的链表中拆下一个区域变为两个2^(k+1)区域，一个挂到编号为k+1的链表上，把另一个拆为两个2^k的区域，一个返还给申请者，把另一个挂到编号为k的链表上。

4、如果k+2的链表也为空，则一直向上迭代，直到编号为10的链表为止，如果编号为10的链表还为空，则申请失败。

向buddy system中释放页面过程：

在向buddy system 释放页面时，总会检测释放的页面和链表中其他页面是否可以组成一个更大一级的页面，如果可以组成，则把这两个区域组成一个并挂到更高一级的链表中。这个过程是迭代的，释放过程会一层层向上找伙伴，然后合并成更大的，再向上找伙伴，实在找不到了就停止了！

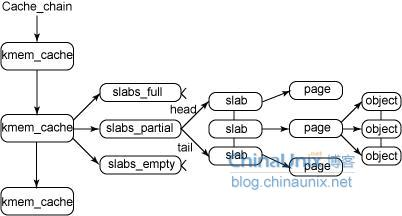
疑问：按照上面的说法，是否会出现这种情况，在释放某个页面导致所有页面都组成了标号为10的连续页面了。等到再需要分配1个页面时，又要一级一级地拆分。这样的话效率是否很低？？

是否在buddy system 每个链表结构中设一个门限值会更好？释放时标记一下可以组成buddy的两个连续区域，只有该级空闲的区域个数超过门限后才组成buddy并挂到上一级链表上。当然，这个门限值可以由内核根据目前总的空闲页面数量进行动态调整。

4、Slab

下图中给出了 slab 结构的高层组织结构。在最高层是 cache\_chain，这是一个 slab 缓存的链接列表。可以用来查找最适合所需要的分配大小的缓存。cache\_chain 的每个元素都是一个 kmem\_cache 结构的引用。一个kmem\_cache中的所有object大小都相同。

slab **分配器的主要结构**

****

slab是基于buddy system的， 每个slab占用一个或多个连续页，即一个buddy链中的1个或多个页面。

每个缓存都包含了一个 **slabs** 列表，这是一段连续的内存块（通常都是页面）。存在 3 种 slab：

**slabs\_full**

完全分配的 slab ，即其维护的空闲object链表为空

**slabs\_partial**

部分分配的 slab

**slabs\_empty**

空 slab，或者没有对象被分配，即其inuse标志位0.

注意 slabs\_empty 列表中的 slab 是进行**回收**的主要备选对象。正是通过此过程，slab 所使用的内存被返回给操作系统供其他用户使用。

slab 列表中的每个 slab 都是一个连续的内存块（从buddy申请的一个或多个连续页），它们被划分成一个个对象，这些对象是分配和释放的基本元素。在slab扩展时或把slab占用的内存块释放到buddy系统时， slab是最小分配单位。通常来说，每个 slab 被分配为多个对象。由于对象是从 slab中进行分配和释放的，因此单个 slab 可以在 slab 列表之间进行移动。例如，当一个 slab 中的所有对象都被使用完时，就从 slabs\_partial 列表中移动到 slabs\_full 列表中。当一个 slab 完全被分配并且有对象被释放后，就从 slabs\_full 列表中移动到 slabs\_partial 列表中。当所有对象都被释放之后，就从 slabs\_partial 列表移动到 slabs\_empty 列表中。

slab 背后的动机

与传统的内存管理模式相比， slab 缓存分配器提供了很多优点。首先，内核通常依赖于对小对象的分配，它们会在系统生命周期内进行无数次分配。slab 缓存分配器通过对类似大小的对象进行缓存而提供这种功能，从而避免了常见的碎片问题。slab 分配器还支持通用对象的初始化，从而避免了为同一目而对一个对象重复进行初始化。最后，slab 分配器还可以支持硬件缓存对齐和着色，这允许不同缓存中的对象占用相同的缓存行，从而提高缓存的利用率并获得更好的性能。

（三）           单个进程的内存管理

每个进程的task\_struct中都有一个active\_mm成员，类型为struct mm\_struct，内核就是利用该成员管理进程虚拟空间的。参见数据结构task\_struct，为了方便阅读，删除了该结构中无关的成员变量。

struct task\_struct{

      struct mm\_struct \*mm, \*active\_mm;

      }

参考下面的数据结构定义。数据结构struct mm\_struct 中的成员mm\_rb指向了一棵红黑树的根，该进程的所有申请的虚拟空间都以起始虚拟地址为红黑树的key值挂到了这棵红黑树上。mm\_struct 中的成员map\_count指示该进程拥有的虚拟空间的个数，pgd指向该进程的页转换表。

struct mm\_struct{

      struct vm\_area\_struct \* mmap; /\* list of VMAs 指向若干个VMA组成的链表 \*/

      struct rb\_root mm\_rb;               指向一棵红黑树

      struct vm\_area\_struct \* mmap\_cache; 指向最近找到的虚拟存储区域

      int map\_count;                  /\* number of VMAs \*/  虚拟区间的个数

      pgd\_t \* pgd;             指向页转换表

}

数据结构struct vm\_area\_struct定义了一个连续的虚拟地址空间，包括起始地址和结束地址，以及红黑树节点vm\_rb。内核就是以vm\_start为key值把vm\_rb挂到进程内存红黑树上的。

struct vm\_area\_struct{

      struct mm\_struct \* vm\_mm;    /\* The address space we belong to. \*/

      unsigned long vm\_start;          /\* Our start address within vm\_mm. \*/

      unsigned long vm\_end; /\* The first byte after our end address within vm\_mm. \*/

      struct rb\_node vm\_rb;      这个虚拟区域对应的红黑树的节点

}

内核在给进程分配了一块虚拟地址内存块后，就将该区域挂接到进程的红黑树上，此时内核尚未给该进程分配实际的内存。在进程访问该区域时则产生缺页中断，在中断中检查访问的区域已经分配给进程后，则分配实际内存页面，并更新该进程的页转换查找表。中断返回，进程重新执行触发中断的指令，并继续运行。

当进程释放一块内存区域后，内核会立即收回分配给该区域的物理内存页面。

（四）           malloc软件

下面内容节选自文章《内存相关分享》，连接为http://www.cppblog.com/CppExplore/archive/2010/03/30/111049.html

应用层面的开发并不是直接调用sbrk/mmap之类的函数，而是调用malloc/free等malloc子系统提供的函数，linux上安装的大多为DougLea的dlmalloc或者其变形ptmalloc。下面以dlmalloc为例说明malloc工作的原理。

**1 dlmalloc下名词解释：**

**boundary tag:** 边界标记，每个空闲内存块均有头部表识和尾部标识，尾部表识的作为是合并空闲内存块时更快。这部分空间属于无法被应用层面使用浪费的内存空间。

**smallbins:** 小内存箱。dlmalloc将8,16,24......512大小的内存分箱，相临箱子中的内存相差8字节。每个箱子中的内存大小均相同，并且以双向链表连接。

**treebins:** 树结构箱。大于512字节的内存不再是每8字节1箱，而是一个范围段一箱。比如512~640, 640~896.....每个箱子的范围段依次是128，256，512......。每箱中的结构不再是双向链表，而是树形结构。

**dv chunk:**  当申请内存而在对应大小的箱中找不到大小合适的内存，则从更大的箱中找一块内存，划分出需要的内存，剩余的内存称之为dv chunk.

**top chunk:** 当dlmalloc中管理的内存都找不到合适的内存时，则调用sbrk从系统申请内存，可以增长内存方向的chunk称为top chunk.

**2 内存分配算法**

        从合适的箱子中寻找内存块-->从相临的箱子中寻找内存块-->从dv chunk分配内存-->从其他可行的箱子中分配内存-->从top chunk中分配内存-->调用sbrk/mmap申请内存

**3 内存释放算法**

       临近内存合并-->如属于top chunk，判断top chunk>128k，是则归还系统

                              -->不属于chunk，则归相应的箱子

dlmalloc还有小内存缓存等其他机制。可以看出经过dlmalloc，频繁调用malloc/free并不会产生内存碎片，只要后续还有相同的内存大小的内存被申请，仍旧会使用以前的合适内存，除非大量调用malloc之后少量释放free，并且新的malloc又大于以前free的内存大小，造成dlmalloc不停的从系统申请内存，而free掉的小内存因被使用的内存割断，而使top chunk<128k，不能归还给系统。即便如此，占用的总内存量也小于的确被使用的内存量的2倍（使用的内存和空闲的内存交叉分割，并且空闲的内存总是小于使用的内存大小）。因此可以说，在没有内存泄露的情况，常规频繁调用malloc/free并不会产生内存碎片。

Linux进程调度

Linux进程调度的目标

　　　　1.高效性：高效意味着在相同的时间下要完成更多的任务。调度程序会被频繁的执行，所以调度程序要尽可能的高效；

　　　　2.加强交互性能:在系统相当的负载下，也要保证系统的响应时间；

　　　　3.保证公平和避免饥渴；

　　　　4.SMP调度：调度程序必须支持多处理系统；

　　　　5.软实时调度：系统必须有效的调用实时进程，但不保证一定满足其要求；

**Linux进程优先级**

　　进程提供了两种优先级，一种是普通的进程优先级，第二个是实时优先级。前者适用SCHED\_NORMAL调度策略，后者可选SCHED\_FIFO或SCHED\_RR调度策略。**任何时候，实时进程的优先级都高于普通进程**，实时进程只会被更高级的实时进程抢占，同级实时进程之间是按照FIFO（一次机会做完）或者RR（多次轮转）规则调度的。

**首先，说下实时进程的调度**

　　实时进程，只有静态优先级，因为内核不会再根据休眠等因素对其静态优先级做调整，其范围在0~MAX\_RT\_PRIO-1间。默认MAX\_RT\_PRIO配置为100，也即，默认的实时优先级范围是0~99。而nice值，影响的是优先级在MAX\_RT\_PRIO~MAX\_RT\_PRIO+40范围内的进程。

　　不同与普通进程，系统调度时，实时优先级高的进程总是先于优先级低的进程执行。知道实时优先级高的实时进程无法执行。实时进程总是被认为处于活动状态。如果有数个 优先级相同的实时进程，那么系统就会按照进程出现在队列上的顺序选择进程。假设当前CPU运行的实时进程A的优先级为a，而此时有个优先级为b的实时进程B进入可运行状态，那么只要b<a，系统将中断A的执行，而优先执行B，直到B无法执行（无论A，B为何种实时进程）。

 　　不同调度策略的实时进程只有在相同优先级时才有可比性：

 　　1. 对于FIFO的进程，意味着只有当前进程执行完毕才会轮到其他进程执行。由此可见相当霸道。

 　　2. 对于RR的进程。一旦时间片消耗完毕，则会将该进程置于队列的末尾，然后运行其他相同优先级的进程，如果没有其他相同优先级的进程，则该进程会继续执行。

 　　总而言之，对于实时进程，高优先级的进程就是大爷。它执行到没法执行了，才轮到低优先级的进程执行。等级制度相当森严啊。

**重头戏，说下非实时进程调度**

**引子**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | 将当前目录下的documents目录打包，但不希望tar占用太多CPU：    nice -19 tar zcf pack.tar.gz documents    这个“-19”中的“-”仅表示参数前缀；所以，如果希望赋予tar进程最高的优先级，则执行：    nice --19 tar zcf pack.tar.gz documents    也可修改已经存在的进程的优先级：    将PID为1799的进程优先级设置为最低：    renice 19 1799    renice命令与nice命令的优先级参数的形式是相反的，直接以优先级值作为参数即可，无“-”前缀说法。 |

**言归正传**

  　　Linux对普通的进程，根据动态优先级进行调度。而动态优先级是由静态优先级（static\_prio）调整而来。Linux下，静态优先级是用户不可见的，隐藏在内核中。而内核提供给用户一个可以影响静态优先级的接口，那就是nice值，两者关系如下：

　　static\_prio=MAX\_RT\_PRIO +nice+ 20

　　nice值的范围是-20~19，因而静态优先级范围在100~139之间。nice数值越大就使得static\_prio越大，最终进程优先级就越低。

　　ps -el 命令执行结果：NI列显示的每个进程的nice值，PRI是进程的优先级（如果是实时进程就是静态优先级，如果是非实时进程，就是动态优先级）

　　而进程的时间片就是完全依赖 static\_prio 定制的，见下图，摘自《深入理解linux内核》，

https://images0.cnblogs.com/i/401155/201404/161210240576893.jpg

 　　我们前面也说了，系统调度时，还会考虑其他因素，因而会计算出一个叫进程动态优先级的东西，根据此来实施调度。因为，不仅要考虑静态优先级，也要考虑进程的属性。例如如果进程属于交互式进程，那么可以适当的调高它的优先级，使得界面反应地更加迅速，从而使用户得到更好的体验。Linux2.6 在这方面有了较大的提高。Linux2.6认为，交互式进程可以从平均睡眠时间这样一个measurement进行判断。进程过去的睡眠时间越多，则越有可能属于交互式进程。则系统调度时，会给该进程更多的奖励（bonus），以便该进程有更多的机会能够执行。奖励（bonus）从0到10不等。

　　系统会严格按照动态优先级高低的顺序安排进程执行。动态优先级高的进程进入非运行状态，或者时间片消耗完毕才会轮到动态优先级较低的进程执行。动态优先级的计算主要考虑两个因素：静态优先级，进程的平均睡眠时间也即bonus。计算公式如下，

     dynamic\_prio = max (100, min (static\_prio - bonus + 5, 139))

　　在调度时，Linux2.6 使用了一个小小的trick，就是算法中经典的空间换时间的思想[**还没对照源码确认**]，使得计算最优进程能够在O(1)的时间内完成。

**为什么根据睡眠和运行时间确定奖惩分数是合理的**

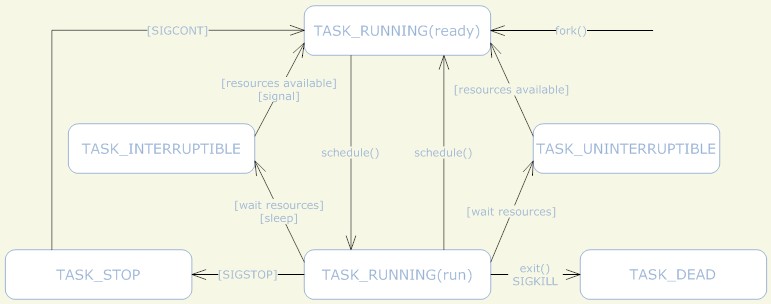
　　睡眠和CPU耗时反应了进程IO密集和CPU密集两大瞬时特点，不同时期，一个进程可能即是CPU密集型也是IO密集型进程。对于表现为IO密集的进程，应该经常运行，但每次时间片不要太长。对于表现为CPU密集的进程，CPU不应该让其经常运行，但每次运行时间片要长。交互进程为例，假如之前其其大部分时间在于等待CPU，这时为了调高相应速度，就需要增加奖励分。另一方面，如果此进程总是耗尽每次分配给它的时间片，为了对其他进程公平，就要增加这个进程的惩罚分数。**可以参考CFS的virtutime机制.**

**现代方法CFS**

　　不再单纯依靠进程优先级绝对值，而是参考其绝对值，综合考虑所有进程的时间，给出当前调度时间单位内其应有的权重，也就是，每个进程的权重X单位时间=应获cpu时间，但是这个应得的cpu时间不应太小（假设阈值为1ms），否则会因为切换得不偿失。但是，当进程足够多时候，肯定有很多不同权重的进程获得相同的时间——最低阈值1ms，所以，CFS只是近似完全公平。

    详情参考 《[linux内核cfs浅析](http://hi.baidu.com/_kouu/item/055bd19af9f6b9dc1f4271d1)》

**Linux进程状态机**

****

　　进程是通过fork系列的系统调用（fork、clone、vfork）来创建的，内核（或内核模块）也可以通过kernel\_thread函数创建内核进程。这些创建子进程的函数本质上都完成了相同的功能——将调用进程复制一份，得到子进程。（可以通过选项参数来决定各种资源是共享、还是私有。）  
那么既然调用进程处于TASK\_RUNNING状态（否则，它若不是正在运行，又怎么进行调用？），则子进程默认也处于TASK\_RUNNING状态。  
另外，在系统调用clone和内核函数kernel\_thread也接受CLONE\_STOPPED选项，从而将子进程的初始状态置为 TASK\_STOPPED。

 　　进程创建后，状态可能发生一系列的变化，直到进程退出。而尽管进程状态有好几种，但是进程状态的变迁却只有两个方向——从TASK\_RUNNING状态变为非TASK\_RUNNING状态、或者从非TASK\_RUNNING状态变为TASK\_RUNNING状态。总之，TASK\_RUNNING是必经之路，不可能两个非RUN状态直接转换。

也就是说，如果给一个TASK\_INTERRUPTIBLE状态的进程发送SIGKILL信号，这个进程将先被唤醒（进入TASK\_RUNNING状态），然后再响应SIGKILL信号而退出（变为TASK\_DEAD状态）。并不会从TASK\_INTERRUPTIBLE状态直接退出。

  　　进程从非TASK\_RUNNING状态变为TASK\_RUNNING状态，是由别的进程（也可能是中断处理程序）执行唤醒操作来实现的。执行唤醒的进程设置被唤醒进程的状态为TASK\_RUNNING，然后将其task\_struct结构加入到某个CPU的可执行队列中。于是被唤醒的进程将有机会被调度执行。

 　　而进程从TASK\_RUNNING状态变为非TASK\_RUNNING状态，则有两种途径：

*1、响应信号而进入TASK\_STOPED状态、或TASK\_DEAD状态；  
　　2、执行系统调用主动进入TASK\_INTERRUPTIBLE状态（如nanosleep系统调用）、或TASK\_DEAD状态（如exit系统调用）；或由于执行系统调用需要的资源得不到满　　　　 足，而进入TASK\_INTERRUPTIBLE状态或TASK\_UNINTERRUPTIBLE状态（如select系统调用）。  
　　显然，这两种情况都只能发生在进程正在CPU上执行的情况下。*

*通过ps命令我们能够查看到系统中存在的进程，以及它们的状态：*

R(TASK\_RUNNING)，可执行状态。

只有在该状态的进程才可能在CPU上运行。而同一时刻可能有多个进程处于可执行状态，这些进程的task\_struct结构（进程控制块）被放入对应CPU的可执行队列中（一个进程最多只能出现在一个CPU的可执行队列中）。进程调度器的任务就是从各个CPU的可执行队列中分别选择一个进程在该CPU上运行。  
只要可执行队列不为空，其对应的CPU就不能偷懒，就要执行其中某个进程。一般称此时的CPU“忙碌”。对应的，CPU“空闲”就是指其对应的可执行队列为空，以致于CPU无事可做。  
有人问，为什么死循环程序会导致CPU占用高呢？因为死循环程序基本上总是处于TASK\_RUNNING状态（进程处于可执行队列中）。除非一些非常极端情况（比如系统内存严重紧缺，导致进程的某些需要使用的页面被换出，并且在页面需要换入时又无法分配到内存……），否则这个进程不会睡眠。所以CPU的可执行队列总是不为空（至少有这么个进程存在），CPU也就不会“空闲”。

很多操作系统教科书将正在CPU上执行的进程定义为RUNNING状态、而将可执行但是尚未被调度执行的进程定义为READY状态，这两种状态在linux下统一为 TASK\_RUNNING状态。

S(TASK\_INTERRUPTIBLE)，可中断的睡眠状态。

处于这个状态的进程因为等待某某事件的发生（比如等待socket连接、等待信号量），而被挂起。这些进程的task\_struct结构被放入对应事件的等待队列中。当这些事件发生时（由外部中断触发、或由其他进程触发），对应的等待队列中的一个或多个进程将被唤醒。

通过ps命令我们会看到，一般情况下，进程列表中的绝大多数进程都处于TASK\_INTERRUPTIBLE状态（除非机器的负载很高）。毕竟CPU就这么一两个，进程动辄几十上百个，如果不是绝大多数进程都在睡眠，CPU又怎么响应得过来。

D(TASK\_UNINTERRUPTIBLE)，不可中断的睡眠状态。

与TASK\_INTERRUPTIBLE状态类似，进程处于睡眠状态，但是此刻进程是不可中断的。不可中断，指的并不是CPU不响应外部硬件的中断，而是指进程不响应异步信号。  
绝大多数情况下，进程处在睡眠状态时，总是应该能够响应异步信号的。否则你将惊奇的发现，kill -9竟然杀不死一个正在睡眠的进程了！于是我们也很好理解，为什么ps命令看到的进程几乎不会出现TASK\_UNINTERRUPTIBLE状态，而总是TASK\_INTERRUPTIBLE状态。

而TASK\_UNINTERRUPTIBLE状态存在的意义就在于，内核的某些处理流程是不能被打断的。如果响应异步信号，程序的执行流程中就会被插入一段用于处理异步信号的流程（这个插入的流程可能只存在于内核态，也可能延伸到用户态），于是原有的流程就被中断了（参见《linux异步信号handle浅析》）。  
在进程对某些硬件进行操作时（比如进程调用read系统调用对某个设备文件进行读操作，而read系统调用最终执行到对应设备驱动的代码，并与对应的物理设备进行交互），可能需要使用TASK\_UNINTERRUPTIBLE状态对进程进行保护，以避免进程与设备交互的过程被打断，造成设备陷入不可控的状态。（比如read系统调用触发了一次磁盘到用户空间的内存的DMA，如果DMA进行过程中，进程由于响应信号而退出了，那么DMA正在访问的内存可能就要被释放了。）这种情况下的TASK\_UNINTERRUPTIBLE状态总是非常短暂的，通过ps命令基本上不可能捕捉到。

linux系统中也存在容易捕捉的TASK\_UNINTERRUPTIBLE状态。执行vfork系统调用后，父进程将进入TASK\_UNINTERRUPTIBLE状态，直到子进程调用exit或exec。  
通过下面的代码就能得到处于TASK\_UNINTERRUPTIBLE状态的进程：  
#include <unistd.h>  
void main() {  
if (!vfork()) sleep(100);  
}  
编译运行，然后ps一下：  
kouu@kouu-one:~/test$ ps -ax | grep a\.out  
4371 pts/0 D+ 0:00 ./a.out  
4372 pts/0 S+ 0:00 ./a.out  
4374 pts/1 S+ 0:00 grep a.out  
然后我们可以试验一下TASK\_UNINTERRUPTIBLE状态的威力。不管kill还是kill -9，这个TASK\_UNINTERRUPTIBLE状态的父进程依然屹立不倒。

T(TASK\_STOPPED or TASK\_TRACED)，暂停状态或跟踪状态。

向进程发送一个SIGSTOP信号，它就会因响应该信号而进入TASK\_STOPPED状态（除非该进程本身处于TASK\_UNINTERRUPTIBLE状态而不响应信号）。（SIGSTOP与SIGKILL信号一样，是非常强制的。不允许用户进程通过signal系列的系统调用重新设置对应的信号处理函数。）  
向进程发送一个SIGCONT信号，可以让其从TASK\_STOPPED状态恢复到TASK\_RUNNING状态。

当进程正在被跟踪时，它处于TASK\_TRACED这个特殊的状态。“正在被跟踪”指的是进程暂停下来，等待跟踪它的进程对它进行操作。比如在gdb中对被跟踪的进程下一个断点，进程在断点处停下来的时候就处于TASK\_TRACED状态。而在其他时候，被跟踪的进程还是处于前面提到的那些状态。  
对于进程本身来说，TASK\_STOPPED和TASK\_TRACED状态很类似，都是表示进程暂停下来。  
而TASK\_TRACED状态相当于在TASK\_STOPPED之上多了一层保护，处于TASK\_TRACED状态的进程不能响应SIGCONT信号而被唤醒。只能等到调试进程通过ptrace系统调用执行PTRACE\_CONT、PTRACE\_DETACH等操作（通过ptrace系统调用的参数指定操作），或调试进程退出，被调试的进程才能恢复TASK\_RUNNING状态。

Z(TASK\_DEAD - EXIT\_ZOMBIE)，退出状态，进程成为僵尸进程。

进程在退出的过程中，处于TASK\_DEAD状态。

在这个退出过程中，进程占有的所有资源将被回收，除了task\_struct结构（以及少数资源）以外。于是进程就只剩下task\_struct这么个空壳，故称为僵尸。  
之所以保留task\_struct，是因为task\_struct里面保存了进程的退出码、以及一些统计信息。而其父进程很可能会关心这些信息。比如在shell中，$?变量就保存了最后一个退出的前台进程的退出码，而这个退出码往往被作为if语句的判断条件。  
当然，内核也可以将这些信息保存在别的地方，而将task\_struct结构释放掉，以节省一些空间。但是使用task\_struct结构更为方便，因为在内核中已经建立了从pid到task\_struct查找关系，还有进程间的父子关系。释放掉task\_struct，则需要建立一些新的数据结构，以便让父进程找到它的子进程的退出信息。

父进程可以通过wait系列的系统调用（如wait4、waitid）来等待某个或某些子进程的退出，并获取它的退出信息。然后wait系列的系统调用会顺便将子进程的尸体（task\_struct）也释放掉。  
子进程在退出的过程中，内核会给其父进程发送一个信号，通知父进程来“收尸”。这个信号默认是SIGCHLD，但是在通过clone系统调用创建子进程时，可以设置这个信号。

通过下面的代码能够制造一个EXIT\_ZOMBIE状态的进程：  
#include <unistd.h>  
void main() {  
if (fork())  
while(1) sleep(100);  
}  
编译运行，然后ps一下：  
kouu@kouu-one:~/test$ ps -ax | grep a\.out  
10410 pts/0 S+ 0:00 ./a.out  
10411 pts/0 Z+ 0:00 [a.out] <defunct>  
10413 pts/1 S+ 0:00 grep a.out

只要父进程不退出，这个僵尸状态的子进程就一直存在。那么如果父进程退出了呢，谁又来给子进程“收尸”？  
当进程退出的时候，会将它的所有子进程都托管给别的进程（使之成为别的进程的子进程）。托管给谁呢？可能是退出进程所在进程组的下一个进程（如果存在的话），或者是1号进程。所以每个进程、每时每刻都有父进程存在。除非它是1号进程。

1号进程，pid为1的进程，又称init进程。  
linux系统启动后，第一个被创建的用户态进程就是init进程。它有两项使命：  
1、执行系统初始化脚本，创建一系列的进程（它们都是init进程的子孙）；  
2、在一个死循环中等待其子进程的退出事件，并调用waitid系统调用来完成“收尸”工作；  
init进程不会被暂停、也不会被杀死（这是由内核来保证的）。它在等待子进程退出的过程中处于TASK\_INTERRUPTIBLE状态，“收尸”过程中则处于TASK\_RUNNING状态。

X(TASK\_DEAD - EXIT\_DEAD)，退出状态，进程即将被销毁。

而进程在退出过程中也可能不会保留它的task\_struct。比如这个进程是多线程程序中被detach过的进程（进程？线程？参见《linux线程浅析》）。或者父进程通过设置SIGCHLD信号的handler为SIG\_IGN，显式的忽略了SIGCHLD信号。（这是posix的规定，尽管子进程的退出信号可以被设置为SIGCHLD以外的其他信号。）  
此时，进程将被置于EXIT\_DEAD退出状态，这意味着接下来的代码立即就会将该进程彻底释放。所以EXIT\_DEAD状态是非常短暂的，几乎不可能通过ps命令捕捉到。

**一些重要的杂项**

**调度程序的效率**  
“优先级”明确了哪个进程应该被调度执行，而调度程序还必须要关心效率问题。调度程序跟内核中的很多过程一样会频繁被执行，如果效率不济就会浪费很多CPU时间，导致系统性能下降。  
在linux 2.4时，可执行状态的进程被挂在一个链表中。每次调度，调度程序需要扫描整个链表，以找出最优的那个进程来运行。复杂度为O(n)；  
在linux 2.6早期，可执行状态的进程被挂在N(N=140)个链表中，每一个链表代表一个优先级，系统中支持多少个优先级就有多少个链表。每次调度，调度程序只需要从第一个不为空的链表中取出位于链表头的进程即可。这样就大大提高了调度程序的效率，复杂度为O(1)；  
在linux 2.6近期的版本中，可执行状态的进程按照优先级顺序被挂在一个红黑树（可以想象成平衡二叉树）中。每次调度，调度程序需要从树中找出优先级最高的进程。复杂度为O(logN)。  
  
那么，为什么从linux 2.6早期到近期linux 2.6版本，调度程序选择进程时的复杂度反而增加了呢？  
这是因为，与此同时，调度程序对公平性的实现从上面提到的第一种思路改变为第二种思路（通过动态调整优先级实现）。而O(1)的算法是基于一组数目不大的链表来实现的，按我的理解，这使得优先级的取值范围很小（区分度很低），不能满足公平性的需求。而使用红黑树则对优先级的取值没有限制（可以用32位、64位、或更多位来表示优先级的值），并且O(logN)的复杂度也还是很高效的。  
  
**调度触发的时机**  
调度的触发主要有如下几种情况：  
1、当前进程（正在CPU上运行的进程）状态变为非可执行状态。  
进程执行系统调用主动变为非可执行状态。比如执行nanosleep进入睡眠、执行exit退出、等等；  
进程请求的资源得不到满足而被迫进入睡眠状态。比如执行read系统调用时，磁盘高速缓存里没有所需要的数据，从而睡眠等待磁盘IO；  
进程响应信号而变为非可执行状态。比如响应SIGSTOP进入暂停状态、响应SIGKILL退出、等等；  
  
2、抢占。进程运行时，非预期地被剥夺CPU的使用权。这又分两种情况：进程用完了时间片、或出现了优先级更高的进程。  
优先级更高的进程受正在CPU上运行的进程的影响而被唤醒。如发送信号主动唤醒，或因为释放互斥对象（如释放锁）而被唤醒；  
内核在响应时钟中断的过程中，发现当前进程的时间片用完；  
内核在响应中断的过程中，发现优先级更高的进程所等待的外部资源的变为可用，从而将其唤醒。比如CPU收到网卡中断，内核处理该中断，发现某个socket可读，于是唤醒正在等待读这个socket的进程；再比如内核在处理时钟中断的过程中，触发了定时器，从而唤醒对应的正在nanosleep系统调用中睡眠的进程；  
  
**内核抢占**  
理想情况下，只要满足“出现了优先级更高的进程”这个条件，当前进程就应该被立刻抢占。但是，就像多线程程序需要用锁来保护临界区资源一样，内核中也存在很多这样的临界区，不大可能随时随地都能接收抢占。  
linux 2.4时的设计就非常简单，内核不支持抢占。进程运行在内核态时（比如正在执行系统调用、正处于异常处理函数中），是不允许抢占的。必须等到返回用户态时才会触发调度（确切的说，是在返回用户态之前，内核会专门检查一下是否需要调度）；  
linux 2.6则实现了内核抢占，但是在很多地方还是为了保护临界区资源而需要临时性的禁用内核抢占。  
  
也有一些地方是出于效率考虑而禁用抢占，比较典型的是spin\_lock。spin\_lock是这样一种锁，如果请求加锁得不到满足（锁已被别的进程占有），则当前进程在一个死循环中不断检测锁的状态，直到锁被释放。  
为什么要这样忙等待呢？因为临界区很小，比如只保护“i+=j++;”这么一句。如果因为加锁失败而形成“睡眠-唤醒”这么个过程，就有些得不偿失了。  
那么既然当前进程忙等待（不睡眠），谁又来释放锁呢？其实已得到锁的进程是运行在另一个CPU上的，并且是禁用了内核抢占的。这个进程不会被其他进程抢占，所以等待锁的进程只有可能运行在别的CPU上。（如果只有一个CPU呢？那么就不可能存在等待锁的进程了。）  
而如果不禁用内核抢占呢？那么得到锁的进程将可能被抢占，于是可能很久都不会释放锁。于是，等待锁的进程可能就不知何年何月得偿所望了。  
  
对于一些实时性要求更高的系统，则不能容忍spin\_lock这样的东西。宁可改用更费劲的“睡眠-唤醒”过程，也不能因为禁用抢占而让更高优先级的进程等待。比如，嵌入式实时linux montavista就是这么干的。  
由此可见，实时并不代表高效。很多时候为了实现“实时”，还是需要对性能做一定让步的。  
  
**多处理器下的负载均衡**  
前面我们并没有专门讨论多处理器对调度程序的影响，其实也没有什么特别的，就是在同一时刻能有多个进程并行地运行而已。那么，为什么会有“多处理器负载均衡”这个事情呢？  
如果系统中只有一个可执行队列，哪个CPU空闲了就去队列中找一个最合适的进程来执行。这样不是很好很均衡吗？  
的确如此，但是多处理器共用一个可执行队列会有一些问题。显然，每个CPU在执行调度程序时都需要把队列锁起来，这会使得调度程序难以并行，可能导致系统性能下降。而如果每个CPU对应一个可执行队列则不存在这样的问题。  
另外，多个可执行队列还有一个好处。这使得一个进程在一段时间内总是在同一个CPU上执行，那么很可能这个CPU的各级cache中都缓存着这个进程的数据，很有利于系统性能的提升。  
所以，在linux下，每个CPU都有着对应的可执行队列，而一个可执行状态的进程在同一时刻只能处于一个可执行队列中。  
  
于是，“多处理器负载均衡”这个麻烦事情就来了。内核需要关注各个CPU可执行队列中的进程数目，在数目不均衡时做出适当调整。什么时候需要调整，以多大力度进程调整，这些都是内核需要关心的。当然，尽量不要调整最好，毕竟调整起来又要耗CPU、又要锁可执行队列，代价还是不小的。  
另外，内核还得关心各个CPU的关系。两个CPU之间，可能是相互独立的、可能是共享cache的、甚至可能是由同一个物理CPU通过超线程技术虚拟出来的……CPU之间的关系也是实现负载均衡的重要依据。关系越紧密，进程在它们之间迁移的代价就越小。参见《[linux内核SMP负载均衡浅析](http://hi.baidu.com/_kouu/blog/item/acc103cf4430d61593457e46.html?timeStamp=1302706107099)》。

**优先级继承**  
由于互斥，一个进程（设为A）可能因为等待进入临界区而睡眠。直到正在占有相应资源的进程（设为B）退出临界区，进程A才被唤醒。  
可能存在这样的情况：A的优先级非常高，B的优先级非常低。B进入了临界区，但是却被其他优先级较高的进程（设为C）抢占了，而得不到运行，也就无法退出临界区。于是A也就无法被唤醒。  
A有着很高的优先级，但是现在却沦落到跟B一起，被优先级并不太高的C抢占，导致执行被推迟。这种现象就叫做优先级反转。  
  
出现这种现象是很不合理的。较好的应对措施是：当A开始等待B退出临界区时，B临时得到A的优先级（还是假设A的优先级高于B），以便顺利完成处理过程，退出临界区。之后B的优先级恢复。这就是优先级继承的方法。  
  
**中断处理线程化**  
在linux下，中断处理程序运行于一个不可调度的上下文中。从CPU响应硬件中断自动跳转到内核设定的中断处理程序去执行，到中断处理程序退出，整个过程是不能被抢占的。  
一个进程如果被抢占了，可以通过保存在它的进程控制块（task\_struct）中的信息，在之后的某个时间恢复它的运行。而中断上下文则没有task\_struct，被抢占了就没法恢复了。  
中断处理程序不能被抢占，也就意味着中断处理程序的“优先级”比任何进程都高（必须等中断处理程序完成了，进程才能被执行）。但是在实际的应用场景中，可能某些实时进程应该得到比中断处理程序更高的优先级。  
于是，一些实时性要求更高的系统就给中断处理程序赋予了task\_struct以及优先级，使得它们在必要的时候能够被高优先级的进程抢占。但是显然，做这些工作是会给系统造成一定开销的，这也是为了实现“实时”而对性能做出的一种让步。

**Linux Scheduling Domains**

Scheduling Domains 引入的背景

Scheduling Domains 是现代硬件技术尤其是多 CPU 多核技术发展的产物。现在，一个复杂的高端系统由上到下可以这样构成：

1. 它是一个 NUMA 架构的系统，系统中的每个 Node 访问系统中不同区域的内存有不同的速度。
2. 同时它又是一个 SMP 系统。由多个物理 CPU(Physical Package) 构成。这些物理 CPU 共享系统中所有的内存。但都有自己独立的 Cache 。
3. 每个物理 CPU 又由多个核 (Core) 构成，即 Multi-core 技术或者叫 Chip-level Multi processor(CMP) 。这些核都被集成在一块 die 里面。一般有自己独立的 L1 Cache，但可能共享 L2 Cache 。
4. 每个核中又通过 SMT 之类的技术实现多个硬件线程，或者叫 Virtual CPU( 比如 Intel 的 Hyper-threading 技术 ) 。这些硬件线程，逻辑上看是就是一个 CPU 。它们之间几乎所有的东西都共享。包括 L1 Cache，甚至是逻辑运算单元 (ALU) 以及 Power 。

在上述系统中，最小的执行单元是逻辑 CPU，进程的调度执行也是相对于逻辑 CPU 的。因此，后文皆简称逻辑 CPU 为 CPU，是物理 CPU 时会特别说明。

在这样复杂的系统，调度器要解决的一个首要问题就是如何发挥这么多 CPU 的性能，使得负载均衡。不存某些 CPU 一直很忙，进程在排队等待运行，而某些 CPU 却是处于空闲状态。

但是在这些 CPU 之间进行 Load Balance 是有代价的，比如对处于两个不同物理 CPU 的进程之间进行负载平衡的话，将会使得 Cache 失效。造成效率的下降。而且过多的 Load Balance 会大量占用 CPU 资源。

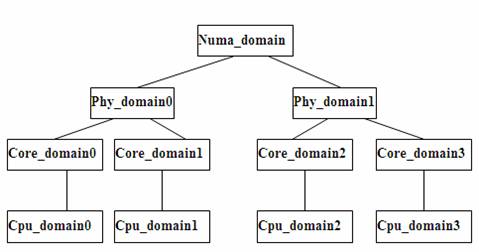
还有一个要考虑的就是功耗 (Power) 的问题。一个物理 CPU 中的两个 Virtual CPU 各执行一个进程，显然比两个物理 CPU 中的 Virtual CPU 各执行一个进程节省功耗。因为硬件上可以实现一个物理 CPU 不用时关掉它以节省功耗。

为了解决上述的这些问题，内核开发人员 Nick Piggin 等人在 Linux 2.6 中引入基于 Scheduling Domains 的解决方案。

**Each CPU has a "base" scheduling domain** (**struct sched\_domain**). The domain hierarchy is built from these base domains via the ->parent pointer. ->parent MUST be NULL terminated, and domain structures should be per-CPU as they are locklessly updated.

**Each scheduling domain spans a number of CPUs** (stored in the ->span field). **A domain's span MUST be a superset of it child's span** (this restriction could be relaxed if the need arises), and a base domain for CPU i MUST span at least i. The top domain for each CPU will generally span all CPUs in the system although strictly it doesn't have to, but this could lead to a case where some CPUs will never be given tasks to run unless the CPUs allowed mask is explicitly set. A sched domain's span means "balance process load among these CPUs".

每个 Scheduling Domain 其实就是具有相同属性的一组 cpu 的集合。并且跟据 Hyper-threading, Multi-core, SMP, NUMA architectures 这样的系统结构划分成不同的级别。不同级之间通过指针链接在一起，从而形成一种的树状的关系。如下图所示：

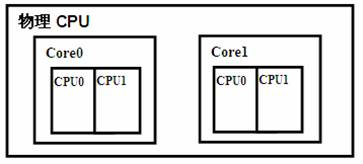


**图 1. Scheduling Domains 原理**

负载平衡就是针对 Scheduling domain 的。从叶节点往上遍历。直到所有的 domain 中的负载都是平衡的。当然对不同的 domain 会有不同的策略识别是否负载不平衡，以及不同的调度策略。通过这样的方式，从而很好的发挥众多 cpu 的效率。

假设每个物理 CPU 只有两个核，每个核只有两个逻辑 CPU 。

如下图所示：

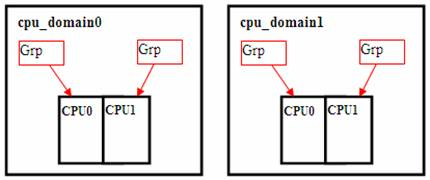


**图 2. 物理 CPU 示意图**

当系统启动时，会分别把每个核的两个逻辑 CPU 放入一个 Scheduling Domain，

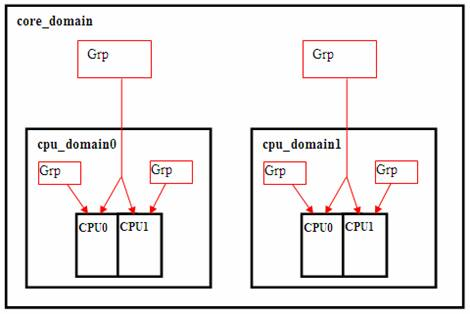
这个级别的 domain 叫做 cpu\_domain 。其中每个 domain 包括两个 CPU groups，每个 CPU group 只有一个逻辑 CPU 。

**图 3. 逻辑 CPU**

****

同时每个物理 CPU 的两个核被放入一个高一级的 Scheduling Domain 。这个 domain 命名为 core\_domain 。其中每个 domain 包括两个 CPU groups，每个 CPU group 有两个逻辑 CPU 。

如下图所示：



**图 4. CPU group**

对于我们前述的复杂系统，再往上的话依次还有 phys\_domain( 物理 CPU 放入的 domain) ；

numa\_domain(NUMA 中的 16 个 nodes 放入的 domain) ； allnode\_domain( 所有 NUMA 中的 node 放入的 domain) 。从而将所有 CPU 组织成一个基于 domain 的层次结构

Scheduling Domains 结构

基于 Scheduling Domains 的调度器引入了一组新的数据结构。下面先讲一下两个主要的数据结构：

* struct sched\_domain: 代表一个 Scheduling Domain，也就是一个 CPU 集合，这个集合里所有的 CPU 都具有相同的属性和调度策略。 Load Balance 是针对每个 domain 里的 CPU 进行的。这里要注意 Scheduling Domains 是分级的。像上节所讲的复杂系统就分为 Allnuma\_domain,Numa\_domain, Phy\_domain, Core\_domain, Smt\_domain(Cpu\_domain) 五个等级。

**表 1. sched\_domain 数据结构**

|  |  |  |
| --- | --- | --- |
| **类型** | **名称** | **描述** |
| struct domain \* | Parent | 当前 domain 的父 domain |
| struct domain \* | Child | 当前 domain 的子 domain |
| struct sched\_group | groups | the balancing groups of the domain |
| cpumask\_t | Span | 当前 domain 中的所有 cpu 位图 |
| unsigned long | min\_interval | 最小的 load balance 间隔 |
| unsigned long | max\_interval | 最大的 load balance 间隔 |
| unsigned int | busy\_factor | Busy 时延迟进行 balance 的系数 |
| unsigned int | busy\_idx |  |
| unsigned int | idle\_idx |  |
| unsigned int | newidle\_idx |  |
| unsigned int | wake\_idx |  |
| unsigned int | forkexec\_idx |  |
| int | flags | 当前 domain 的一些状态标记 |
| enum sched\_domain\_level | level | 当前 domain 的级别 |
| unsigned long | last\_balance | 当前 domain 最近一次进行 balance 时的时间 (jiffies 为单位 ) |
| unsigned int | balance\_interval | 进行 balance 的时间间隔（ms 为单位） |
| unsigned int | nr\_balance\_failed | balance 失败的次数 |

**Each scheduling domain must have one or more CPU groups** (struct sched\_group) which are organised as a circular one way linked list from the ->groups pointer. **The union of cpumasks of these groups MUST be the same as the domain's span**. The intersection of cpumasks from any two of these groups MUST be the empty set. The group pointed to by the ->groups pointer MUST contain the CPU to which the domain belongs. Groups may be shared among CPUs as they contain read only data after they have been set up.

**Balancing within a sched domain occurs between groups**. That is, each group is treated as one entity. The load of a group is defined as the sum of the load of each of its member CPUs, and **only when the load of a group becomes out of balance are tasks moved between groups**.

**表 2. sched\_group 数据结构**

|  |  |  |
| --- | --- | --- |
| **类型** | **名称** | **描述** |
| struct sched\_group \* | next | 下一个 group 的指针 |
| cpumask\_t | cpumask | 当前 group 有哪些 CPU |
| unsigned int | \_\_cpu\_power | 当前 group 的 CPU power |
| u32 | reciprocal\_cpu\_power | CPU power 的倒数 |

In kernel/sched/core.c, **trigger\_load\_balance()** is run periodically on each CPU through scheduler\_tick(). It raises a **softirq** after the next regularly scheduled rebalancing event for the current runqueue has arrived. The actual load balancing workhorse, **run\_rebalance\_domains**()-**>rebalance\_domains**(), is then run in softirq context (SCHED\_SOFTIRQ).

The latter function takes two arguments: the current CPU and whether it was idle at the time the **scheduler\_tick()** happened and iterates over all sched domains our CPU is on, **starting from its base domain and going up the ->parent chain.** While doing that, it checks to see if the current domain has exhausted its rebalance interval. If so, it runs **load\_balance() on that domain**. It then checks the parent sched\_domain (if it exists), and the parent of the parent and so forth.

Initially, **load\_balance() finds the busiest group in the current sched domain**. If it succeeds, it looks for the busiest runqueue of all the CPUs' runqueues in that group. If it manages to find such a runqueue, it locks both our initial CPU's runqueue and the newly found busiest one and starts moving tasks from it to our runqueue. The exact number of tasks amounts to an imbalance previously computed while iterating over this sched domain's groups.

\*\*\* Implementing sched domains \*\*\* The "base" domain will "span" the first level of the hierarchy. In the case of SMT, **you'll span all siblings of the physical CPU, with each group being a single virtual CPU**.逻辑CPU

**In SMP, the parent of the base domain will span all physical CPUs in the node**. Each group being a single physical CPU. Then with NUMA, the parent of the SMP domain will span the entire machine, with each group having the cpumask of a node. Or, you could do multi-level NUMA or Opteron, for example, might have just one domain covering its one NUMA level.物理CPU

The implementor should read comments in include/linux/sched.h: struct sched\_domain fields, SD\_FLAG\_\*, SD\_\*\_INIT to get an idea of the specifics and what to tune.

Architectures may retain the regular override the default SD\_\*\_INIT flags while using the generic domain builder in kernel/sched/core.c if they wish to retain the **traditional SMT->SMP->NUMA topology** (or some subset of that). This can be done by #define'ing ARCH\_HASH\_SCHED\_TUNE.

Alternatively, the architecture may completely override the generic domain builder by #define'ing ARCH\_HASH\_SCHED\_DOMAIN, and exporting your arch\_init\_sched\_domains function. This function will attach domains to all CPUs using cpu\_attach\_domain.

The sched-domains debugging infrastructure can be enabled by enabling CONFIG\_SCHED\_DEBUG. This enables an error checking parse of the sched domains which should catch most possible errors (described above). It also prints out the domain structure in a visual format.

Scheduling Domains 实现

对每个 Scheduling Domain 中的 CPU groups 之间的 CPU 进行的。

每个 Scheduling Domain 都包含一些重要的信息用来决定在这级 domain 的 CPU groups 之间如何进行 Load Balance 。

* 典型的一些原则如下：
  1. 在 cpu\_domain 级： 因为是**共享 cache**，cpu\_power 也基本是共用的。所以可以在这个 domain 级的 cpu groups 之间可以不受限制的进行 load balance 。
  2. 在 core\_domain 级：可能会**共享 L2 级 cache**, 具体跟实现相关了。因此这一级的 balance 相对没那么频繁。要 core 之间负载的不平衡达到一定程度才进行 balance 。
  3. 在 phys\_domain 级：在这一个 domain 级，如果进行 balance 。则每个物理 CPU 上的 Cache 会失效一段时间，并且要考虑 cpu\_power，因为物理 CPU 级的 power 一般是被数关系。比如两个物理 CPU 是 power\*2，而不像 core, 或逻辑 CPU，只是 power\*1.1 这样的关系。
  4. 在 numa\_domain 级：这一级的开销最大，一般很少进行 balance 。
* 基本实现
* Linux 主要通过以下几个方面来实现基于 Scheduling domains 的 Load Balance 。
  1. 周期性的 load balance
  2. 每次时钟中断到来，如果发现当前 cpu 的运行队列需要进行下一次的 balance 的时间已
  3. 经到了，则触发 SCHED\_SOFTIRQ 软中断。
  4. **触发软中断**
  5. if (time\_after\_eq(jiffies, rq->next\_balance))

|  |
| --- |
| raise\_softirq(SCHED\_SOFTIRQ); |

* 1. 软中断的执行函数是 run\_rebalance\_domains(), 它主要是再调 rebalance\_domains() 来实现。
  2. **清单 1. run\_rebalance\_domains()**

|  |
| --- |
| static void run\_rebalance\_domains(struct softirq\_action \*h)    {  int this\_cpu = smp\_processor\_id();  struct rq \*this\_rq = cpu\_rq(this\_cpu);  enum cpu\_idle\_type idle = this\_rq->idle\_at\_tick ?  CPU\_IDLE : CPU\_NOT\_IDLE;  rebalance\_domains(this\_cpu, idle);  #ifdef CONFIG\_NO\_HZ  #endif    } |

* 1. rebalance\_domains()，根据 domain 的级别，从下往上扫描每一级 Scheduling Domain 。如果发现这个 domain 的 balance 之间的间隔时间到了，则进一步进行 task 的移动。不同级别的 domain 是会有不同的间隔时间的。而且级别越高值越大，因为移动 task 的代价越大。
  2. **清单 2. rebalance\_domains()**

|  |
| --- |
| static void rebalance\_domains(int cpu, enum cpu\_idle\_type idle)    {  int balance = 1;  struct rq \*rq = cpu\_rq(cpu);  unsigned long interval;  struct sched\_domain \*sd;  /\* Earliest time when we have to do rebalance again \*/  unsigned long next\_balance = jiffies + 60\*HZ;  int update\_next\_balance = 0;  int need\_serialize;  cpumask\_t tmp;  for\_each\_domain(cpu, sd) {  if (!(sd->flags & SD\_LOAD\_BALANCE))  continue;  interval = sd->balance\_interval;  if (idle != CPU\_IDLE)  interval \*= sd->busy\_factor;  /\* scale ms to jiffies \*/  interval = msecs\_to\_jiffies(interval);  if (unlikely(!interval))  interval = 1;  if (interval > HZ\*NR\_CPUS/10)  interval = HZ\*NR\_CPUS/10;  need\_serialize = sd->flags & SD\_SERIALIZE;  if (need\_serialize) {  if (!spin\_trylock(&balancing))  goto out;  }  if (time\_after\_eq(jiffies, sd->last\_balance + interval)) {  if (load\_balance(cpu, rq, sd, idle, &balance, &tmp)) {  /\*  \* We've pulled tasks over so either we're no  \* longer idle, or one of our SMT siblings is  \* not idle.  \*/  idle = CPU\_NOT\_IDLE;  }  sd->last\_balance = jiffies;  }  if (need\_serialize)  spin\_unlock(&balancing);    out:  if (time\_after(next\_balance, sd->last\_balance + interval)) {  next\_balance = sd->last\_balance + interval;  update\_next\_balance = 1;  }  /\*  \* Stop the load balance at this level. There is another  \* CPU in our sched group which is doing load balancing more  \* actively.  \*/  if (!balance)  break;  }  /\*  \* next\_balance will be updated only when there is a need.  \* When the cpu is attached to null domain for ex, it will not be  \* updated.  \*/  if (likely(update\_next\_balance))  rq->next\_balance = next\_balance;    } |

* 1. 针对 CPU IDLE 的处理
  2. 如果一个逻辑 CPU 进入了 IDLE 状态，并且它所属的 domain 设置了 SD\_BALANCE\_NEWIDLE，则马上就会进行 balance，把忙的 CPU 上的进程 move 过来，从而最大的发挥多 CPU 的优势。
  3. 针对 fork(), exec() 的处理
  4. 当一个进行调用 exec() 执行时，本来就是要加载一个新进程，缓存本来就会失效。所以，move 到哪里都可以。因此找设置了 SD\_BALANCE\_EXEC 标记的 domain 。然后把进程移动到那个 domain 中最闲的 CPU group 的 CPU 上。
  5. fork() 时也进行类似的处理。
  6. 其它因素比如针对 cpu\_power 的处理