# 深入理解Linux内核架构

1. 简介和概述
   1. 内核的任务

内核是硬件与软件之间的一个中间层，其将应用程序的请求传递给硬件，并充当底层驱动程序，对系统中的各种设备和组件进行寻址。

应用程序角度——内核可被认为是一台增强的计算机。若干程序并发运行——内核就是一个资源管理器。内核也可被视为库，提供一组面向系统的命令。

* 1. 实现策略

内核由两种范型:微内核和宏内核。

微内核：只有最基本的功能直接由中央内核实现。所有其他的功能委托给一些独立

进程，这些进程通过明确定义的通信接口与中心内核通信。

宏内核:宏内核是构建系统内核的传统部分，内核的全部代码如内存管理、文件系统、设备驱动程序都被打包到一个文件中。

1.3内核的组成部分：

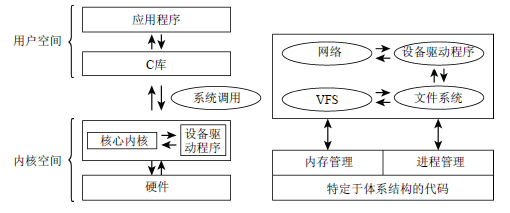


图1-3.1 Linux内核的层次概述和完整Linux系统中的各个层次

1.3.1 进程、进程切换、调度

传统上UNIX操作系统下运行的应用程序、服务器及其他程序都成为进程。

进程是内存分配的基本单位，各个进程的地址空间是完全独立的。当CPU数目大于进程数目时，各个运行的进程才可以说是并行执行的，否则需要进行通过进程的切换来应对这种情况，同时这也造成了CPU可同时处理很多个程序的假象。进城之间的切换称之为进程切换。

内核对进程运行时长进行调度的策略称之为调度。

1.3.2 UNIX进程

Linux中每个进程都依赖于一个父进程，内核启动init程序作为第一个进程，该进程负责进一步的系统初始化操作，并显示线路提示符或图形登陆界面。

Linux内核创建进程的两种机制：fork和exec。fork可以创建一个进程的副本，父子进程之间只有PID不同。Linux使用了写时复制技术，其主要原理是将内存复制操作延迟到父进程或子进程向内存中写入数据，在只读情况下两者共用同一内存页。exec采取将新程序加载到当前进程内存中并执行，旧程序被刷出，从而开始执行新程序。

线程是一种轻量级进程，其是任务调度的基本单位。

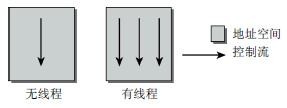


图1.3.2 有线程和无线程的进程对比

命名空间在内核2.6中内集成，因此不同的进程可以看到不同的系统视图，现在比较流行的SDN技术就是基于Linux内核的命名空间机制实现的。每个命名空间都可以包含一个特定的PID集合，或可以提供文件系统的不同视图，在某个命名空间中挂载的文件系统也不会传播到其他命名空间中了。

1.3.3 地址空间和特权级别

计算机是对于内存区域是通过指针寻址，因此CPU的字长决定了其所能管理的地址空间的最大长度。因此32位系统是232=4GB，而64位系统可以管理264B的长度。地址空间的最大长度又称虚拟地址空间，其与实际可用物理内存无关。

Linux将虚拟地址划分为两个部分：内核空间和用户空间。如下图1.3.3所示，系统中每个用户都有自己的虚拟地址范围，从0到TASK\_SIZE。用户空间之上的部分为内核空间，用户禁止访问。在IA-32系统中，地址在3GB处划分，用户占用3GB，内核空间有1G可用。但是这个值也是可以修改的。

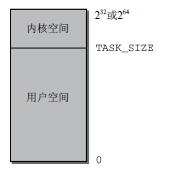


图1.3.3 .1 虚拟地址空间划分

1）特权级别

内核将虚拟地址空间进行了划分，使各个进程能够相互隔离以达到保护的作用。基本上目前所有现代CPU都提供了特权级别的划分。如图1.3.3.2所示，IA-32划分了四种特权级别，Linux只有两种状态:核心态和用户态。其中用户态禁止访问内核空间，用户进程不能对内核空间的数据进行读写操作，亦无法执行其中代码。该机制可防止进程无意间修改彼此数据而造成的干扰。

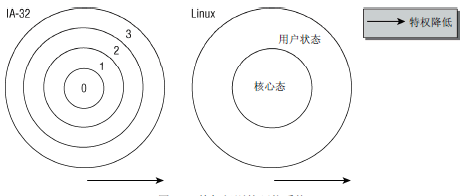


图1.3.3.2 特权级别的环状系统

从用户态切换到内核态是通过系统调用的方式来实现的，内核会检测进程是否允许执行想要的操作，然后代表进程执行所需操作后返回用户态。当然内核也可由硬件终端激活，然后中断上下文中运行的进程，但是不能访问虚拟地址中用户空间部分。

1. 虚拟地址和物理地址

CPU分配内存实际分配的虚拟内存，因此需要将实际可用物理内存映射成虚拟地址.Linux是采用页表来将物理地址映射成虚拟地址，这个映射关系到进程的用户空间和内核空间，而物理地址则用来寻址。如图1.3.3.3所示。

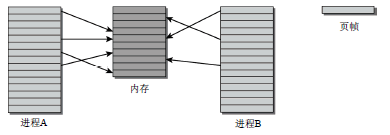


图1.3.3.3 虚拟地址和物理地址

物理内存页经常被称为页帧，而页则专指虚拟地址空间中的页。

1.3.4 页表

页表，即将虚拟地址空间映射到物理地址空间的数据结构。在IA-32体系结构中采用的是4KB页，由于在4GB内存前提下64位系统中每个进程中都要自身的页表，导致所有内存都来保存页表，这个极大的浪费了资源。因此在Linux中采用了四级页表进行优化，在这里我们以三级页表进行描述，如图1.3.4.1所示。

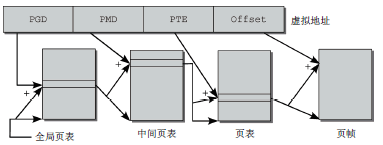


图1.3.4.1 分配虚拟地址

PGD:全局页目录，其主要用于索引进程中一个数组，其数组项主要用来指向中间页目录的起始地址。

PMD:中间页目录，其账户要用来索引PMD，其数组项也是指针，指向下一级数组——页表。

PTE：页表数组，用作页表的索引，虚拟内存和页帧之间的映射就此完成，因此页表的数组指向页帧。

虚拟地址最后的一部分为页内部偏移量，其指向页内部的一个字节位置。

多级页表相对于数目相比节省了内存，但是访问内存需要进行逐级访问，客采用下面方法进行优化。

1. MMU（内存管理单元）：CPU中一个专门的部分，其优化了内存的访问操作。
2. 将地址转换中最频繁出现的那些地址保存到称为地址转换后备缓冲器（TLB）的CPU高速缓存中。无需访问内存中的页表直接从高速缓存中读取地址数据。

1.3.5 物理内存的分配

在内核分配内存时，必须记录页帧的已分配内存或空闲状态，以免两个进程使用同样的内存区域。内核只分配完整页帧以保证操作尽快完成，而用户标准库则将内存划分为更小的部分，并为进程分配内存。

1. 伙伴系统

伙伴系统是一种可以分配连续页，快速检测内存页中的连续区域的技术。伙伴系统中的空闲内存块总是两两分组的额，两个彼此为伙伴，互相独立，若两个伙伴彼此独立，则可合并为更大的快，如下图1.3.5所示。

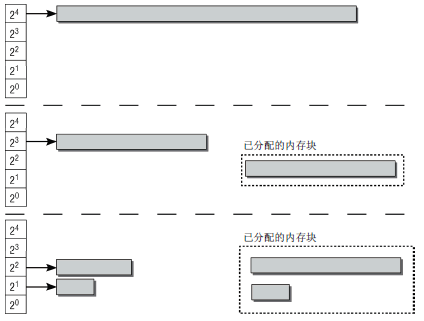
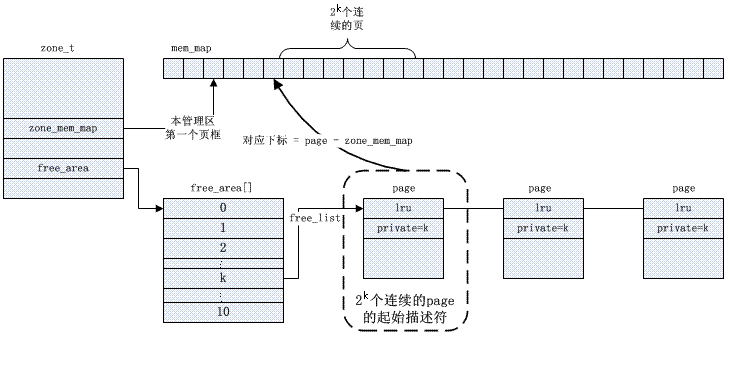


图1.3.5.1 伙伴系统示意图

内核将所有相同伙伴都放置到同一个列表中管理，若需要8个页帧，则将16个页

帧组成的伙伴系统拆分，一块进行分配，剩余的则放置到8页对应内存块列表。若下一个请求2个连续页帧，则8页分裂成两个伙伴，每个包含4个页帧。其中一块放回伙伴列表，另一块再次分裂，每个包含两页，一个回系统，另一个满足应用程序。



1.3.5.2 伙伴系统

2）slab缓存

slab缓存弥补了slab系统中只能分配完整页帧的内存块的缺点，slab缓存有两种方法分配内存。

1. 对频繁使用对象，内核定义了只包含所需类型对象实例的缓存，slab缓存自动维护与伙伴系统的维护，在缓存用尽时会请求新的页帧。
2. 对通常情况下小内存块的分配，内核可针对不同大小的对象定义一组slab缓存，用户可以使用函数去访问这些内存，这组函数名称前面都增加了前缀k，表明其与内核相关联，如kmalloc和kfree。

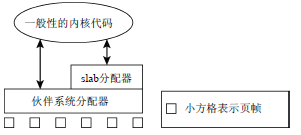


图1.3.5 伙伴系统和slab分配器合作进行内存分配

3）页面交换和页面回收

页面 交换是一种利用磁盘空间作为扩展内存，从而扩大了可用内存。

页面回收则用于将内存映射被修改的内容与底层的块设备同步，亦称数据回写。

1.3.6 计时

计时：内核测量时间以及不同时间点的时差的功能。jiffies是一个合适的时间坐标，其是一种全局变量，其数值通常介于100和1000之间。计时周期可以动态改变

1.3.7 系统调用

系统调用是用户进程与内核交互的经典方法。其主要分为以下几组：

进程管理：进程创建与销毁，查询与调试。

信号：发送信号、定时器以及相关处理机制

文件：创建、打开、关闭文件，以及文件读写，查询文件信息和状态等。

目录：创建、删除和重命名目录，查询信息，链接，变更目录等。

保护机制：读取、修改UID/GID等。

定时器函数：定时器函数与统计信息。

1.3.8 设备驱动程序、块设备和字符设备

外设可分为以下两类：

1. 字符设备：提供连续的数据流，应用程序可顺序读取。
2. 块设备：应用程序可随机访问设备数据，程序可自行确定读取数据的位置。

1.3.9 网络

网卡可以通过设别驱动程序控制，内核中属于特殊情况。因为网卡不能利用设备文件访问，原因在于网路通信期间，数据打包到了各种协议层，从而是得数据在网络中传播。套接字可以看做是应用程序、文件接口、内核的网络实现之间的代理。

1.3.10 文件系统

文件系统使用目录结构组织存储的数据，并将其他元信息与实际数据关联起来。Linux支持的文件系统有：Ext2、Ext3、ReiserFS、XFS、VFAT，当然还有其他文件系统。其中Ext2基于inode，inode是每个文件的单独管理结构，存储到磁盘上，其数据包含了指向目录下所有文件的inode指针。

VFS:其将各种底层文件系统的具体特性与应用层隔离，其是向上和向下接口的中间层。如下图1.3.10所示。

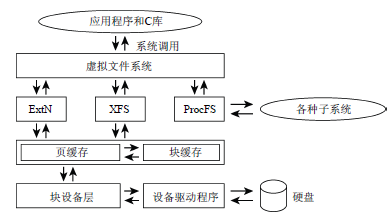


图1.3.10 虚拟文件系统层、文件系统实现和块设备之间的互操作。

1.3.11 模块和热插拔

模块用于在运行动态地向内核添加功能。

模块对于支持热插拔是必需的，某些总线允许在系统运行时连接设别，而无需系统重启。在系统检测到新设备时，其就会加载对应的模块，可以将必要的驱动程序加载到内核。

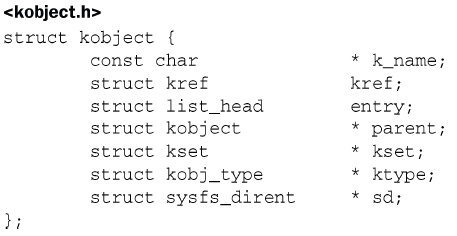
1.3.12 缓存

缓存是一种内核改进系统性能的技术，其可以分为页缓存和块缓存，页缓存是基于页存映射来实现访问块设别，块缓存用于缓存没有组织成页的数据。

1.3.13 对象管理和引用计数

一般性的内核对象机制可用于执行以下对象操作：

1. 引用计数；
2. 管理对象链表；
3. 集合加锁；
4. 将对象属性导出到用户空间。
5. 一般性的内核对象



kobject是通过直接嵌入其他数据结构和其他数据结构连接起来的。

k\_name:对象的文本名称

kref:用于简化引用计数的管理。

entry:用于将一个若干个kobjetc放置到一个链表中。

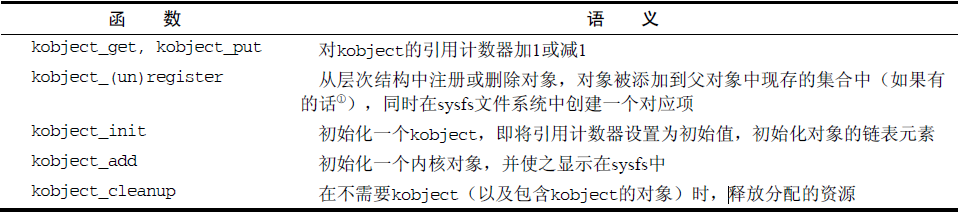
kset:将对象与其他对象放置到一个集合中。

parent:一个指向父对象的指针，可用于在kobject之间建立层次结构。

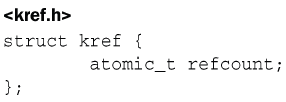
ktype：提供了包含kobject的数据结构的更多信息。

表1.3.11中列出了了kobject的标准方法：

表1.3.11



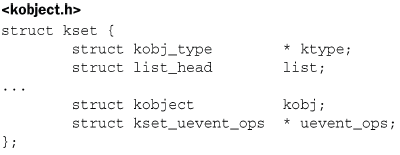
用于管理引用计数的kref结构如下



recount是一个原子数据类型，给出了内核中当前使用某个对象的计数。当计数器达到0时，则说明不需要该对象，可从内存中删除。

1. 对象集合

用到的数据结构定义如下：

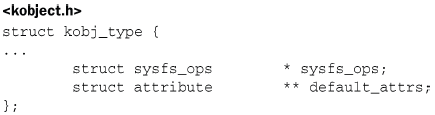


上述集合中的kset只是用来管理kset对象本身。其成员定义如下：

ktype：指向kset中各个内核对象公用的kobj\_type结构。

list：所有属于当前集合内核对象的链表。

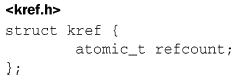
uevent\_ops：用于将集合的状态信息传递给用户层。



上述结构用于描述内核对象的共同特性。

3．引用计数

引用计数：检测内核中对对象的引用次数。被引用一次，引用计数加1,,若不再需要则减1，直至到0。若不再需要，则释放该对象。处理引用计数的结构如下：



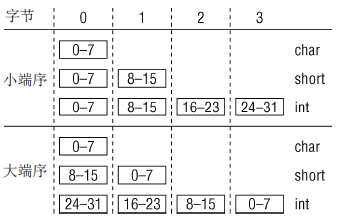
1.3.14 数据类型

1．类型定义

内核使用typedef来定义各种数据类型，以避免依赖于体系结构相关特性。若变量类型来自于typedef，那么就不能直接访问，需要通过辅助函数。

2．字节序

现代计算机字节序分为大端序和小端序。大端序中最高有效字节存储在最低地址，小端格式中，最低有效字节存储在最低地址。如下图1.3.15所示。



1.3.15 大端序和小端序

二．进程管理和调度

2.1 进程优先级

进程分为实时进程和非实时进程。

硬实时进程有严格的时间限制，某些任务必须在指定的时限内完成。软实时进程是硬实时进程的一种弱化形式。大多数进程没有特定时间约束的普通进程。CPU时间分配简图如2.1所示。

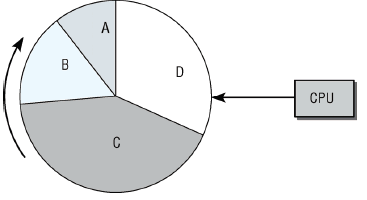


图2.1 时间片分配CPU时间

抢占式多任务处理：各个进程都分配到一定时间段可以移植，当时间到期后，内核

会从进程强制手机控制权，被抢占进程的运行时环境都会被保存起来因此其执行结果不会丢失，恢复时进程环境可完全恢复。

2.2 进程生命周期

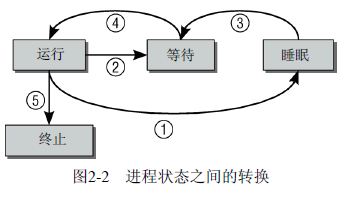
进程可能有以下几种状态：运行、等待和睡眠等。

运行：该进程此刻正在运行。

等待：进程能够运行，但没有得到允许，有时也称就绪态。

睡眠：进程正在睡眠无法运行，其在等待一个外部事件唤醒。

进程的状态转换如图2.2所示。



上文中未列出一种进程的特殊状态——僵尸态。这种进程状态产生的原因：该进程的资源已经释放，但是进程表中仍然有其对应表项。即第一程序必须由另一个进程或用户杀死，第二进程的父进程在子进程终止时未调用wait4函数在子进程终止前终止退出。

抢占式多任务处理

Linux进程管理结构还有两种进程状态选项：用户态和核心态。用户通常都处于用户态，只能访问自身数据，无法干扰其他程序。用户处于核心态即可访问系统数据个功能。

内核的抢占调度模型层次结构：

1. 普通进程总是可能被抢占，甚至是由其他进程抢占。
2. 如果系统处于核心态并正在处理系统调用，那么系统中的其他进程是无法夺取其CPU时间片段的。
3. 中断可以暂停处于用户状态和核心态的进程，中断具有最高优先级。

2.3.1 进程表示

Linux内核及及进程和顺序的所有算法都围绕一个名为task\_struct的数据结构，其位于include/sched.h中。

Linux中task\_struct用来控制管理进程，结构如下：

struct task\_struct   
{

 volatile long state;   /\*说明了该进程是否可以执行,还是可中断等信息。-1表示不可执行，0表示可运行，>0表示禁止\*/   
unsigned long flags;   /\* Flage 是进程号,在调用fork()时给出 \*/  
 int sigpending;   /\* 进程上是否有待处理的信号 \* /

//进程地址空间,区分内核进程与普通进程在内存存放的位置不同  
 mm\_segment\_t addr\_limit; /\* 0-0xBFFFFFFF for user-thead    
 0-0xFFFFFFFF for kernel-thread \*/

 volatile long need\_resched; /\* 调度标志,表示该进程是否需要重新调度,若非0,则当从内核态返回到用户态,会发生调度 \*/  
 int lock\_depth;    /\* 大内核锁深度 \*/   
 long nice;    /\*进程的基本时间片 \*/

 unsigned long policy; /\*进程的调度策略,有三种,实时进程:SCHED\_FIFO,SCHED\_RR, 分时进程:SCHED\_OTHER  
\*/

 struct mm\_struct \*mm;   /\* 进程内存管理信息 \*/  
  
   
 int processor; /\*若进程不在任何CPU上运行, cpus\_runnable 的值是0，否则是1 这个值在运行队列被锁时更新\*/  
 unsigned long cpus\_runnable, cpus\_allowed;  
 struct list\_head run\_list; /\*指向运行队列的指针 \*/  
 unsigned long sleep\_time;  /\* 进程的睡眠时间 \*/  
 struct task\_struct \*next\_task, \*prev\_task; /\*用于将系统中所有的进程连成一个双向循环链表, 其根是init\_task \*/  
 struct mm\_struct \*active\_mm;  
 struct list\_head local\_pages;       /\* 指向本地页面  \*/      
 unsigned int allocation\_order, nr\_local\_pages;  
 struct linux\_binfmt \*binfmt;  /\* 进程所运行的可执行文件的格式 \*/  
 int exit\_code, exit\_signal;  
 int pdeath\_signal;     /\* 父进程终止是向子进程发送的信号 \*/  
 unsigned long personality;  
 /\* Linux可以运行由其他UNIX操作系统生成的符合iBCS2标准的程序 \*/  
 int did\_exec:1;   
 pid\_t pid;    /\* 进程标识符,用来代表一个进程 \*/  
 pid\_t pgrp;   /\*进程组标识,表示进程所属的进程组\*/

 pid\_t tty\_old\_pgrp;  /\* 进程控制终端所在的组标识 \*/  
 pid\_t session;  /\*进程的会话标识 \*/

pid\_t tgid;   
 int leader;     /\*表示进程是否为会话主管\*/   
 struct task\_struct \*p\_opptr,\*p\_pptr,\*p\_cptr,\*p\_ysptr,\*p\_osptr;  
 struct list\_head thread\_group;   /\*线程链表 \*/

struct task\_struct \*pidhash\_next; /\* 用于将进程链入HASH表 \*/  
 struct task\_struct \*\*pidhash\_pprev;  
 wait\_queue\_head\_t wait\_chldexit;  /\* 供wait4()使用 \*/  
 struct completion \*vfork\_done;  /\* 供vfork() 使用 \*/  
 unsigned long rt\_priority; /\* 实时优先级，用它计算实时进程调度时的weight值 \*/  
 /\* it\_real\_value，it\_real\_incr用于REAL定时器，单位为jiffies, 系统根据it\_real\_value设置定时器的第一个终止时间. 在定时器到期时，向进程发送SIGALRM信号，同时根据it\_real\_incr重置终止时间，it\_prof\_value，it\_prof\_incr用于Profile定时器，单位为jiffies。当进程运行时，不管在何种状态下，每个tick都使it\_prof\_value值减一，当减到0时，向进程发送信号SIGPROF，并根据it\_prof\_incr重置时间.it\_virt\_value，it\_virt\_value用于Virtual定时器，单位为jiffies。当进程运行时，不管在何种状态下，每个tick都使it\_virt\_value值减一当减到0时，向进程发送信号SIGVTALRM，根据it\_virt\_incr重置初值。\*/

unsigned long it\_real\_value, it\_prof\_value, it\_virt\_value;  
unsigned long it\_real\_incr, it\_prof\_incr, it\_virt\_value;  
struct timer\_list real\_timer;   /\* 指向实时定时器的指针 \*/  
struct tms times;      /\* 记录进程消耗的时间 \*/  
unsigned long start\_time;  /\* 进程创建的时间 \*/

 long per\_cpu\_utime[NR\_CPUS], per\_cpu\_stime[NR\_CPUS];  /\*记录进程在每个CPU上所消耗的用户态时间和核心态时间 \*/

//内存缺页和交换信息:

/\* min\_flt, maj\_flt累计进程的次缺页数（Copy on　Write页和匿名页）和主缺页数（从映射文件或交换设备读入的页面数）；

nswap记录进程累计换出的页面数，即写到交换设备上的页面数。  
cmin\_flt, cmaj\_flt, cnswap记录本进程为祖先的所有子孙进程的累计次缺页数，主缺页数和换出页面数。

在父进程回收终止的子进程时，父进程会将子进程的这些信息累计到自己结构的这些域中 \*/  
 unsigned long min\_flt, maj\_flt, nswap, cmin\_flt, cmaj\_flt, cnswap;  
 int swappable:1; /\*表示进程的虚拟地址空间是否允许换出\*/

 //进程认证信息  
uid,gid为运行该进程的用户的用户标识符和组标识符，通常是进程创建者的uid，gid

euid，egid为有效uid,gid  
fsuid，fsgid为文件系统uid,gid，这两个ID号通常与有效uid,gid相等，在检查对于文件\*/  
 //suid，sgid为备份uid,gid  
 uid\_t uid,euid,suid,fsuid;  
 gid\_t gid,egid,sgid,fsgid;  
 int ngroups; //记录进程在多少个用户组中  
 gid\_t groups[NGROUPS]; //记录进程所在的组

 //进程的权能，分别是有效位集合，继承位集合，允许位集合  
 kernel\_cap\_t cap\_effective, cap\_inheritable, cap\_permitted;

 int keep\_capabilities:1;  
 struct user\_struct \*user;  
 struct rlimit rlim[RLIM\_NLIMITS];  //与进程相关的资源限制信息  
 unsigned short used\_math;   //是否使用FPU  
 char comm[16];   //进程正在运行的可执行文件名

 //文件系统信息  
 int link\_count, total\_link\_count; /\* 连接数目\*/

 struct tty\_struct \*tty;  /\* NULL if no tty 进程所在的控制终端，如果不需要控制终端，则该指针为空 \*/  
 unsigned int locks;

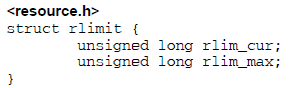
 //进程间通信信息  
 struct sem\_undo \*semundo;  /\* 进程在信号灯上的所有undo操作 \*/  
 struct sem\_queue \*semsleeping; /\* 当进程因为信号灯操作而挂起时，他在该队列中记录等待的操作 \*/  
 struct thread\_struct thread; /\*进程的CPU状态，切换时，要保存到停止进程的task\_struct中 \*/  
 struct fs\_struct \*fs; /\*文件系统信息\*/  
 struct files\_struct \*files; /\*打开文件信息\*/  
 spinlock\_t sigmask\_lock;   /\* 信号处理函数 \*/  
 struct signal\_struct \*sig; //信号处理函数  
 sigset\_t blocked;  //进程当前要阻塞的信号，每个信号对应一位  
 struct sigpending pending;  /\* 进程上是否有待处理的信号 \*/  
 unsigned long sas\_ss\_sp;  
 size\_t sas\_ss\_size;  
 int (\*notifier)(void \*priv);  
 void \*notifier\_data;  
 sigset\_t \*notifier\_mask;

//线程组跟踪

u32 parent\_exec\_id;  
 u32 self\_exec\_id;

spinlock\_t alloc\_lock;  
 void \*journal\_info; /\* 虚拟内存状态\*/  
 };

Linux提供资源限制机制，其使用了task\_struc t的rlim数组，数组项类型为struct rlimit。

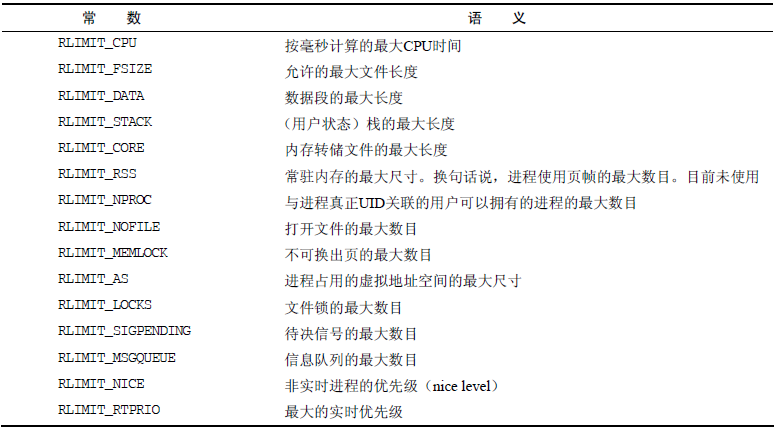


rlim\_cur:进程当前的资源限制，即软限制。

rlim\_max:该限制的最大容许值，即硬限制。

setrlimit和getrlimits用于增减和检查当前限制。表2.3列出了可能的常数及其含义。

表2.3



2.3.2 命名空间

**1.概念**

传统上Linux及其衍生版的UNIX变体中，许多资源都是全局管理的。例如进程PID和用户的UID等全局ID。为了节约成本并且能够保证用户之间的权限不受影响，命名空间提供了一种与KVMhe VMare不同的解决方案。内核通过命名空间将全局资源进行抽象，使得各个进程组分别放到不同的容器，彼此隔离，但是可以允许提供一些接口使得其可以相互通信来降低容器间的间隔。

命名空间给系统建立了不同的视图，如图2.3.2所示。

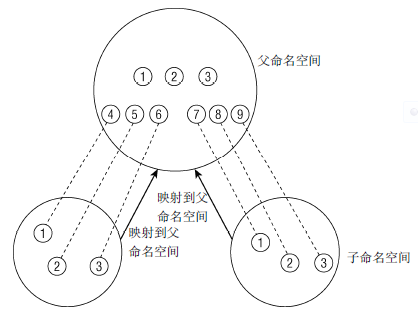


图2.3.2 .1命名空间使得系统呈现不同视图

新的命名空间的创建方法：

1. 在用fork和clone系统调用创建进程时，可以选特定的选项控制是否与父进程共享命名空间。
2. unshare系统调用可将某些部分，如命名空间等从父进程分离。

2.命名空间的实现

命名空间的实现需要两个部分：每个子系统的命名空间结构；将给定进程关联到所属各个命名空间的机制。如图2.3.3.2所示

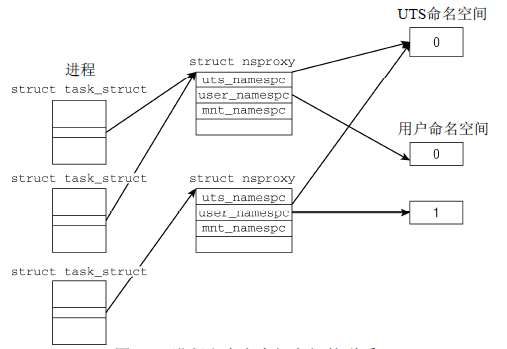
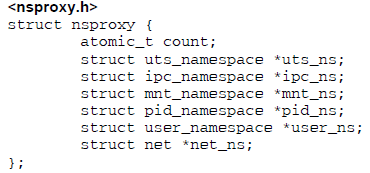


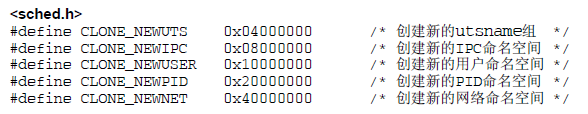
图2.3.3.2 进程和命名空间之间的联系

struct nsproxy结构体汇集了指向特定于子系统的命名空间包装器的指针。其具体成员结构如下：

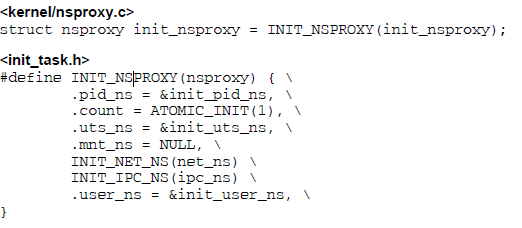


**2.实现**

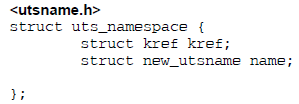
每个进程都关联到了自身的命名空间视图，每个命名空间都有一个对应的标志：



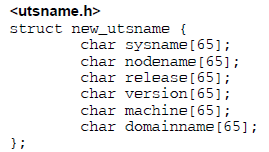
默认命名空间的作用所有属性都相当于全局的。init\_nsproxy定义了初始的全局命名空间，其中维护了指向各子系统初始的命名空间对象的指针。



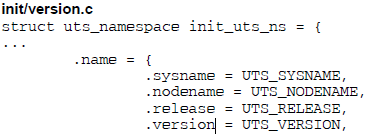
UTS命名空间所有相关信息都汇集到下列结构中的一个实例中：

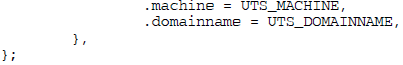


其中kref是一个嵌入的引用计数器，用于对使用struct uts\_namespace的实例进行跟踪，name包含了uts\_namespace的属性信息。struct newname的结构如下所示。



初始设置保存在init\_uts\_ns中：

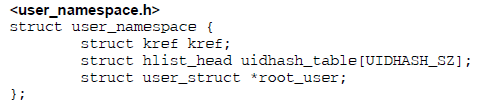




内核创建一个UTS命名空间，是通过copy\_utsname函数实现的，在某个进程调用fork()并通过CLONE\_NEWUTS标志指定创建新的UTS命名空间时 ，则调用该函数。

**用户命名空间**

用户命名空间在要求创建新的用户命名空间时，则生成当前用户命名空间的一份副本，并关联到当前进程的nsproxy实例。



kref：引用计数器。

uidhash\_table：每个struct user\_struct的实例可通过uidhash\_table访问其资源消耗情况。

root\_user：指向root用户的命名空间的指针。

root用户命名空间clone实现：

kernel/user\_namespace.c

static struct　　user\_namespace　　\*clone\_user\_ns(struct user\_namespace　　\*old\_ns)

{

struct user\_namespace　　\*ns;

struct　user\_struct　　\*new\_user;

ns = kmalloc(sizeof(struct user\_namespace) , GFP\_KERNEL);

ns->root\_user　　=　　alloc\_uid(ns, 0);

new\_user　　= alloc\_uid(ns, current->uid);

switch\_uid(new\_user);

return ns;

}

alloc\_uid()：对当前命名空间中给定一个用户，若无对应user\_struct实例，则分配一个新实例。

switch\_uid()：确保从现在开始将新的user\_struct实例用于资源统计，即将struct task\_struct的user成员指向新的user\_struct实例。

2.3.3. 进程ID号

**1）进程ID**

进程ID，简称PID，是进程在命名空间的唯一表示。

线程组ID：每个进程除了PID这个特征值外，还有线程组ID(TGID)，若没有使用线程，则PID和TGID相同。

会话ID（SID）:几个进程可合并成一个会话，会话中所有进程都有同样的SID保存于task\_struct的session成员中，可使用setsid系统调用设置。

进程组ID(PGRP)：独立进程可合并成进程组，进程组内成员进程组ID都相同。

**全局ID和局部ID**

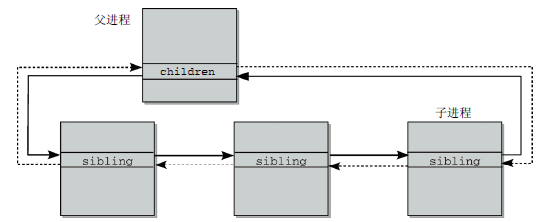
全局ID是内核本身和初始命名空间中的唯一ID号，早系统启动时开始的init进程就属于初始命名空间。

局部ID:局部ID属于某个特定的命名空间，不具备全局有效性。

1. **管理PID**

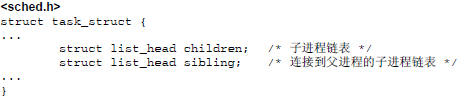
2.3.4 进程关系

如果进程A分支形成进程B，进程A称之为父进程，而进程B则是A进程的子进程。若B再次创建分支进程C，则A和C是祖孙关系。若进程A创建若干分支B1、B2、B3....、Bn，则各个Bi进程之间为兄弟关系。如下图2.3.4说明了可能存在的进程家族关系。



：图2.3.4 进程之间的家族关系

task\_struct数据结构提供了两个链表表头实现了上述关系



children:链表表头，该链表保存了有进程的所有子进程。

sibling:用于将兄弟进程彼此连接起来。

2.4 进程管理相关的系统调用

2.4.1 进程复制

1. fork()是一个重量级调用，因为它建立了一个父进程的完整副本，然后作为子进程执行。Linux后来采用了写时复制技术对其进行了优化。
2. vfork()类似于fork()，但是其父子进程共享数据。vfork()设计用于子进程形成后立即执行execve()加载新程序的情况。子进程退出或开始新程序之前，内核保证父进程处于阻塞状态。
3. clone()产生线程，其可对父子进程之间的共享。复制进程精确控制。

**写时复制**

传统处理方法缺点：

1. 使用了大量内存
2. 复制操作耗费很长时间

写时复制技术（COW）：进程只是复制页表，，从而建立虚拟地址和物理内存页间的关系。因此，fork()之后父子进程的地址空间指向同样的物理内存页，但是父子进程不能修改彼此的页，否则会产生缺页异常。COW机制使得内核尽可能延迟内存页的复制，从而节省时间。

**系统调用**

fork、vfork、clone系统调用的入口点分别是sys\_fork、sys\_vfork和sys\_clone等。上述函数的任务都是从处理器寄存器中提取由用户空间提供的信息，调用与体系结构无关的\_fork函数，do\_fork()负责进程复制。其实现如下：

do\_fork()的实现流程如下图2.4.1所示。

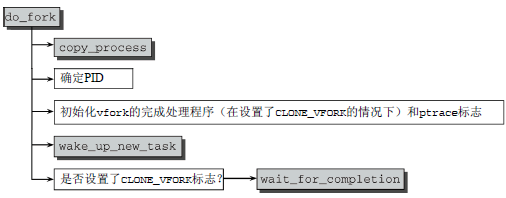


图2.4.1 do\_fork()的代码 流程

/\*\*

\* 负责处理clone,fork,vfork系统调用。clone\_flags-与clone的flag参数相同，stack\_start-与clone的child\_stack相同，regs-指向通用寄存器的值，是在从用户态切换到内核态时被保存到内核态堆栈中的。 stack\_size-未使用,总是为0。 parent\_tidptr,child\_tidptr-clone中对应参数ptid,ctid相同

\*/

long do\_fork(unsigned long clone\_flags,

unsigned long stack\_start,

struct pt\_regs \*regs,

unsigned long stack\_size,

int \_\_user \*parent\_tidptr,

int \_\_user \*child\_tidptr)

{

/\*

clone\_flags:一个标志集合，用来指定控制复制过程的一些属性。最低字节指定了在子进程终止时被发给父进程的信号号码。其余高位字节保存了各种常数。

stack\_start:用户态下栈的起始地址。

regs:一个指向寄存器集合的指针。

stack\_size:用户态的栈的大小。

parent\_tidptr和child\_tidptr:指向用户空间地址的两个指针，分别指向父子进程的PID。

\*/

struct task\_struct \*p;

int trace = 0;

/\*\*

\* 通过查找pidmap\_array位图,为子进程分配新的pid参数.

\*/

long pid = alloc\_pidmap();

if (pid < 0)

return -EAGAIN;

/\*\*

\* 如果父进程正在被跟踪,就检查debugger程序是否想跟踪子进程.并且子进程不是内核进程(CLONE\_UNTRACED未设置)

\* 那么就设置CLONE\_PTRACE标志.

\*/

if (unlikely(current->ptrace)) {

trace = fork\_traceflag (clone\_flags);

if (trace)

clone\_flags |= CLONE\_PTRACE;

}

/\*\*

\* copy\_process复制进程描述符.如果所有必须的资源都是可用的,该函数返回刚创建的task\_struct描述符的地址.

\* 这是创建进程的关键步骤.

\*/

p = copy\_process(clone\_flags, stack\_start, regs, stack\_size, parent\_tidptr, child\_tidptr, pid);

/\*

\* Do this prior waking up the new thread - the thread pointer

\* might get invalid after that point, if the thread exits quickly.

\*/

if (!IS\_ERR(p)) {

struct completion vfork;

if (clone\_flags & CLONE\_VFORK) {

p->vfork\_done = &vfork;

init\_completion(&vfork);

}

/\*\*

\* 如果设置了CLONE\_STOPPED,或者必须跟踪子进程.

\* 就设置子进程为TASK\_STOPPED状态,并发送SIGSTOP信号挂起它.

\*/

if ((p->ptrace & PT\_PTRACED) || (clone\_flags & CLONE\_STOPPED)) {

/\*

\* We'll start up with an immediate SIGSTOP.

\*/

sigaddset(&p->pending.signal, SIGSTOP);

set\_tsk\_thread\_flag(p, TIF\_SIGPENDING);

}

/\*\*

\* 没有设置CLONE\_STOPPED,就调用wake\_up\_new\_task

\* 它调整父进程和子进程的调度参数.

\* 如果父子进程运行在同一个CPU上,并且不能共享同一组页表(CLONE\_VM标志被清0).那么,就把子进程插入父进程运行队列.

\* 并且子进程插在父进程之前.这样做的目的是:如果子进程在创建之后执行新程序,就可以避免写时复制机制执行不必要时页面复制.

\* 否则,如果运行在不同的CPU上,或者父子进程共享同一组页表.就把子进程插入父进程运行队列的队尾.

\*/

if (!(clone\_flags & CLONE\_STOPPED))

wake\_up\_new\_task(p, clone\_flags);

else/\*如果CLONE\_STOPPED标志被设置，就把子进程设置为TASK\_STOPPED状态。\*/

p->state = TASK\_STOPPED;

/\*\*

\* 如果进程正被跟踪,则把子进程的PID插入到父进程的ptrace\_message,并调用ptrace\_notify

\* ptrace\_notify使当前进程停止运行,并向当前进程的父进程发送SIGCHLD信号.子进程的祖父进程是跟踪父进程的debugger进程.

\* dubugger进程可以通过ptrace\_message获得被创建子进程的PID.

\*/

if (unlikely (trace)) {

current->ptrace\_message = pid;

ptrace\_notify ((trace << 8) | SIGTRAP);

}

/\*\*

\* 如果设置了CLONE\_VFORK,就把父进程插入等待队列,并挂起父进程直到子进程结束或者执行了新的程序.

\*/

if (clone\_flags & CLONE\_VFORK) {

wait\_for\_completion(&vfork);

if (unlikely (current->ptrace & PT\_TRACE\_VFORK\_DONE))

ptrace\_notify ((PTRACE\_EVENT\_VFORK\_DONE << 8) | SIGTRAP);

}

} else {

free\_pidmap(pid);

pid = PTR\_ERR(p);

}

return pid;

}

**复制进程**

do\_fork()大多数工作都是由copy\_process完成的，其代码流程如下图2.4.2所示。

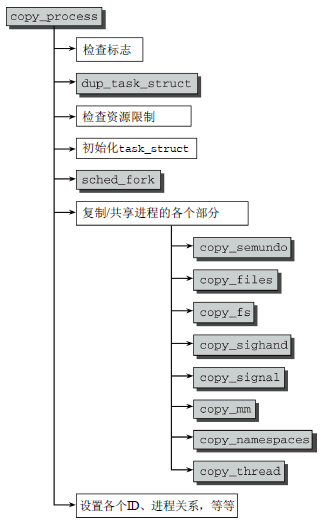


图2.4.2 copy\_process流程图

其实现代码如下：

static task\_t \*copy\_process(unsigned long clone\_flags,

unsigned long stack\_start,

struct pt\_regs \*regs,

unsigned long stack\_size,

int \_\_user \*parent\_tidptr,

int \_\_user \*child\_tidptr,

int pid)

{

int retval;

struct task\_struct \*p = NULL;

/\*\*

\* 检查clone\_flags所传标志的一致性。

\*/

/\*\*

\* 如果CLONE\_NEWNS和CLONE\_FS标志都被设置，返回错误

\*/

if ((clone\_flags & (CLONE\_NEWNS|CLONE\_FS)) == (CLONE\_NEWNS|CLONE\_FS))

return ERR\_PTR(-EINVAL);

/\*

\* Thread groups must share signals as well, and detached threads

\* can only be started up within the thread group.

\*/

/\*\*

\* CLONE\_THREAD标志被设置，并且CLONE\_SIGHAND没有设置。

\* (同一线程组中的轻量级进程必须共享信号)

\*/

if ((clone\_flags & CLONE\_THREAD) && !(clone\_flags & CLONE\_SIGHAND))

return ERR\_PTR(-EINVAL);

/\*

\* Shared signal handlers imply shared VM. By way of the above,

\* thread groups also imply shared VM. Blocking this case allows

\* for various simplifications in other code.

\*/

/\*\*

\* CLONE\_SIGHAND被设置，但是CLONE\_VM没有设置。

\* (共享信号处理程序的轻量级进程也必须共享内存描述符)

\*/

if ((clone\_flags & CLONE\_SIGHAND) && !(clone\_flags & CLONE\_VM))

return ERR\_PTR(-EINVAL);

/\*\*

\* 通过调用security\_task\_create以及稍后调用security\_task\_alloc执行所有附加的安全检查。

\* LINUX2.6提供扩展安全性的钩子函数，与传统unix相比，它具有更加强壮的安全模型。

\*/

retval = security\_task\_create(clone\_flags);

if (retval)

goto fork\_out;

retval = -ENOMEM;

/\*\*

\* 调用dup\_task\_struct为子进程获取进程描述符。

\*/

p = dup\_task\_struct(current);

if (!p)

goto fork\_out;

/\*\*

\* 检查存放在current->sigal->rlim[RLIMIT\_NPROC].rlim\_cur中的限制值，是否小于或者等于用户所拥有的进程数。

\* 如果是，则返回错误码。当然，有root权限除外。

\* p->user表示进程的拥有者，p->user->processes表示进程拥有者当前进程数

\* xie.baoyou注：此处比较是用>=而不是>

\*/

retval = -EAGAIN;

if (atomic\_read(&p->user->processes) >=

p->signal->rlim[RLIMIT\_NPROC].rlim\_cur) {

/\*\*

\* 当然，用户有root权限就另当别论了

\*/

if (!capable(CAP\_SYS\_ADMIN) && !capable(CAP\_SYS\_RESOURCE) &&

p->user != &root\_user)

goto bad\_fork\_free;

}

/\*\*

\* 递增user结构的使用计数器

\*/

atomic\_inc(&p->user->\_\_count);

/\*\*

\* 增加用户拥有的进程计数。

\*/

atomic\_inc(&p->user->processes);

get\_group\_info(p->group\_info);

/\*

\* If multiple threads are within copy\_process(), then this check

\* triggers too late. This doesn't hurt, the check is only there

\* to stop root fork bombs.

\*/

/\*\*

\* 检查系统中的进程数量（nr\_threads）是否超过max\_threads

\* max\_threads的缺省值是由系统内存容量决定的。总的原则是：所有的thread\_info描述符和内核栈所占用的空间

\* 不能超过物理内存的1/8。不过，系统管理可以通过写/proc/sys/kernel/thread-max文件来改变这个值。

\*/

if (nr\_threads >= max\_threads)

goto bad\_fork\_cleanup\_count;

/\*\*

\* 如果新进程的执行域和可招待格式的内核函数都包含在内核中模块中，

\* 就递增它们的使用计数器。

\*/

if (!try\_module\_get(p->thread\_info->exec\_domain->module))

goto bad\_fork\_cleanup\_count;

if (p->binfmt && !try\_module\_get(p->binfmt->module))

goto bad\_fork\_cleanup\_put\_domain;

/\*\*

\* 设置几个与进程状态相关的关键字段。

\*/

/\*\*

\* did\_exec是进程发出的execve系统调用的次数，初始为0

\*/

p->did\_exec = 0;

/\*\*

\* 更新从父进程复制到tsk\_flags字段中的一些标志。

\* 首先清除PF\_SUPERPRIV。该标志表示进程是否使用了某种超级用户权限。

\* 然后设置PF\_FORKNOEXEC标志。它表示子进程还没有发出execve系统调用。

\*/

copy\_flags(clone\_flags, p);

/\*\*

\* 保存新进程的pid值。

\*/

p->pid = pid;

retval = -EFAULT;

/\*\*

\* 如果CLONE\_PARENT\_SETTID标志被设置，就将子进程的PID复制到参数parent\_tidptr指向的用户态变量中。

\* xie.baoyou:想想我们常常调用的pid = fork()语句吧。

\*/

if (clone\_flags & CLONE\_PARENT\_SETTID)

if (put\_user(p->pid, parent\_tidptr))

goto bad\_fork\_cleanup;

p->proc\_dentry = NULL;

/\*\*

\* 初始化子进程描述符中的list\_head数据结构和自旋锁。

\* 并为挂起信号，定时器及时间统计表相关的几个字段赋初值。

\*/

INIT\_LIST\_HEAD(&p->children);

INIT\_LIST\_HEAD(&p->sibling);

p->vfork\_done = NULL;

spin\_lock\_init(&p->alloc\_lock);

spin\_lock\_init(&p->proc\_lock);

clear\_tsk\_thread\_flag(p, TIF\_SIGPENDING);

init\_sigpending(&p->pending);

p->it\_real\_value = 0;

p->it\_real\_incr = 0;

p->it\_virt\_value = cputime\_zero;

p->it\_virt\_incr = cputime\_zero;

p->it\_prof\_value = cputime\_zero;

p->it\_prof\_incr = cputime\_zero;

init\_timer(&p->real\_timer);

p->real\_timer.data = (unsigned long) p;

p->utime = cputime\_zero;

p->stime = cputime\_zero;

p->rchar = 0; /\* I/O counter: bytes read \*/

p->wchar = 0; /\* I/O counter: bytes written \*/

p->syscr = 0; /\* I/O counter: read syscalls \*/

p->syscw = 0; /\* I/O counter: write syscalls \*/

acct\_clear\_integrals(p);

/\*\*

\* 把大内核锁计数器初始化为-1

\*/

p->lock\_depth = -1; /\* -1 = no lock \*/

do\_posix\_clock\_monotonic\_gettime(&p->start\_time);

p->security = NULL;

p->io\_context = NULL;

p->io\_wait = NULL;

p->audit\_context = NULL;

#ifdef CONFIG\_NUMA

p->mempolicy = mpol\_copy(p->mempolicy);

if (IS\_ERR(p->mempolicy)) {

retval = PTR\_ERR(p->mempolicy);

p->mempolicy = NULL;

goto bad\_fork\_cleanup;

}

#endif

p->tgid = p->pid;

if (clone\_flags & CLONE\_THREAD)

p->tgid = current->tgid;

if ((retval = security\_task\_alloc(p)))

goto bad\_fork\_cleanup\_policy;

if ((retval = audit\_alloc(p)))

goto bad\_fork\_cleanup\_security;

/\* copy all the process information \*/

/\*\*

\* copy\_semundo，copy\_files，copy\_fs，copy\_sighand，copy\_signal

\* copy\_mm，copy\_keys，copy\_namespace创建新的数据结构，并把父进程相应数据结构的值复制到新数据结构中。

\* 除非clone\_flags参数指出它们有不同的值。

\*/

if ((retval = copy\_semundo(clone\_flags, p)))

goto bad\_fork\_cleanup\_audit;

if ((retval = copy\_files(clone\_flags, p)))

goto bad\_fork\_cleanup\_semundo;

if ((retval = copy\_fs(clone\_flags, p)))

goto bad\_fork\_cleanup\_files;

if ((retval = copy\_sighand(clone\_flags, p)))

goto bad\_fork\_cleanup\_fs;

if ((retval = copy\_signal(clone\_flags, p)))

goto bad\_fork\_cleanup\_sighand;

if ((retval = copy\_mm(clone\_flags, p)))

goto bad\_fork\_cleanup\_signal;

if ((retval = copy\_keys(clone\_flags, p)))

goto bad\_fork\_cleanup\_mm;

if ((retval = copy\_namespace(clone\_flags, p)))

goto bad\_fork\_cleanup\_keys;

/\*\*

\* 调用copy\_thread，用发出clone系统调用时CPU寄存器的值（它们保存在父进程的内核栈中）

\* 来初始化子进程的内核栈。不过，copy\_thread把eax寄存器对应字段的值（这是fork和clone系统调用在子进程中的返回值）

\* 强行置为0。子进程描述符的thread.esp字段初始化为子进程内核栈的基地址。ret\_from\_fork的地址存放在thread.eip中。

\* 如果父进程使用IO权限位图。则子进程获取该位图的一个拷贝。

\* 最后，如果CLONE\_SETTLS标志被置位，则子进程获取由CLONE系统调用的参数tls指向的用户态数据结构所表示的TLS段。

\*/

retval = copy\_thread(0, clone\_flags, stack\_start, stack\_size, p, regs);

if (retval)

goto bad\_fork\_cleanup\_namespace;

/\*\*

\* 如果clone\_flags参数的值被置为CLONE\_CHILD\_SETTID或CLONE\_CHILD\_CLEARTID

\* 就把child\_tidptr参数的值分别复制到set\_child\_tid或clear\_child\_tid字段。

\* 这些标志说明：必须改变子进程用户态地址空间的dhild\_tidptr所指向的变量的值

\* 不过实际的写操作要稍后再执行。

\*/

p->set\_child\_tid = (clone\_flags & CLONE\_CHILD\_SETTID) ? child\_tidptr : NULL;

/\*

\* Clear TID on mm\_release()?

\*/

p->clear\_child\_tid = (clone\_flags & CLONE\_CHILD\_CLEARTID) ? child\_tidptr: NULL;

/\*

\* Syscall tracing should be turned off in the child regardless

\* of CLONE\_PTRACE.

\*/

/\*\*

\* 清除TIF\_SYSCALL\_TRACE标志。使ret\_from\_fork函数不会把系统调用结束的消息通知给调试进程。

\* 也不应该通知给调试进程，因为子进程并没有调用fork.

\*/

clear\_tsk\_thread\_flag(p, TIF\_SYSCALL\_TRACE);

/\* Our parent execution domain becomes current domain

These must match for thread signalling to apply \*/

p->parent\_exec\_id = p->self\_exec\_id;

/\* ok, now we should be set up.. \*/

/\*\*

\* 用clone\_flags参数低位的信号数据编码统建始化tsk\_exit\_signal字段。

\* 如CLONE\_THREAD标志被置位，就把exit\_signal字段初始化为-1。

\* 这样做是因为：当创建线程时，即使被创建的线程死亡，都不应该给领头进程的父进程发送信号。

\* 而应该是领头进程死亡后，才向其领头进程的父进程发送信号。

\*/

p->exit\_signal = (clone\_flags & CLONE\_THREAD) ? -1 : (clone\_flags & CSIGNAL);

p->pdeath\_signal = 0;

p->exit\_state = 0;

/\* Perform scheduler related setup \*/

/\*\*

\* 调用sched\_fork完成对新进程调度程序数据结构的初始化。

\* 该函数把新进程的状态置为TASK\_RUNNING，并把thread\_info结构的preempt\_count字段设置为1，

\* 从而禁止抢占。

\* 此外，为了保证公平调度，父子进程共享父进程的时间片。

\*/

sched\_fork(p);

/\*

\* Ok, make it visible to the rest of the system.

\* We dont wake it up yet.

\*/

p->group\_leader = p;

INIT\_LIST\_HEAD(&p->ptrace\_children);

INIT\_LIST\_HEAD(&p->ptrace\_list);

/\* Need tasklist lock for parent etc handling! \*/

write\_lock\_irq(&tasklist\_lock);

/\*

\* The task hasn't been attached yet, so cpus\_allowed mask cannot

\* have changed. The cpus\_allowed mask of the parent may have

\* changed after it was copied first time, and it may then move to

\* another CPU - so we re-copy it here and set the child's CPU to

\* the parent's CPU. This avoids alot of nasty races.

\*/

p->cpus\_allowed = current->cpus\_allowed;

/\*\*

\* 初始化子线程的cpu字段。

\*/

set\_task\_cpu(p, smp\_processor\_id());

/\*

\* Check for pending SIGKILL! The new thread should not be allowed

\* to slip out of an OOM kill. (or normal SIGKILL.)

\*/

if (sigismember(&current->pending.signal, SIGKILL)) {

write\_unlock\_irq(&tasklist\_lock);

retval = -EINTR;

goto bad\_fork\_cleanup\_namespace;

}

/\* CLONE\_PARENT re-uses the old parent \*/

/\*\*

\* 初始化表示亲子关系的字段，如果CLONE\_PARENT或者CLONE\_THREAD被设置了

\* 就用current->real\_parent初始化，否则，当前进程就是初创建进程的父进程。

\*/

if (clone\_flags & (CLONE\_PARENT|CLONE\_THREAD))

p->real\_parent = current->real\_parent;

else

p->real\_parent = current;

p->parent = p->real\_parent;

if (clone\_flags & CLONE\_THREAD) {

spin\_lock(&current->sighand->siglock);

/\*

\* Important: if an exit-all has been started then

\* do not create this new thread - the whole thread

\* group is supposed to exit anyway.

\*/

if (current->signal->flags & SIGNAL\_GROUP\_EXIT) {

spin\_unlock(&current->sighand->siglock);

write\_unlock\_irq(&tasklist\_lock);

retval = -EAGAIN;

goto bad\_fork\_cleanup\_namespace;

}

p->group\_leader = current->group\_leader;

if (current->signal->group\_stop\_count > 0) {

/\*

\* There is an all-stop in progress for the group.

\* We ourselves will stop as soon as we check signals.

\* Make the new thread part of that group stop too.

\*/

current->signal->group\_stop\_count++;

set\_tsk\_thread\_flag(p, TIF\_SIGPENDING);

}

spin\_unlock(&current->sighand->siglock);

}

/\*\*

\* 把新进程加入到进程链表

\*/

SET\_LINKS(p);

/\*\*

\* PT\_PTRACED表示子进程必须被跟踪，就把current->parent赋给tsk->parent，并将子进程插入调试程序的跟踪链表中。

\*/

if (unlikely(p->ptrace & PT\_PTRACED))

\_\_ptrace\_link(p, current->parent);

/\*\*

\* 把新进程描述符的PID插入pidhash散列表中。

\*/

attach\_pid(p, PIDTYPE\_PID, p->pid);

attach\_pid(p, PIDTYPE\_TGID, p->tgid);

/\*\*

\* 如果子进程是线程组的领头进程(CLONE\_THREAD标志被清0)

\*/

if (thread\_group\_leader(p)) {

/\*\*

\* 将进程插入相应的散列表。

\*/

attach\_pid(p, PIDTYPE\_PGID, process\_group(p));

attach\_pid(p, PIDTYPE\_SID, p->signal->session);

if (p->pid)

\_\_get\_cpu\_var(process\_counts)++;

}

/\*\*

\* 计数

\*/

nr\_threads++;

total\_forks++;

write\_unlock\_irq(&tasklist\_lock);

retval = 0;

fork\_out:

if (retval)

return ERR\_PTR(retval);

return p;

bad\_fork\_cleanup\_namespace:

exit\_namespace(p);

bad\_fork\_cleanup\_keys:

exit\_keys(p);

bad\_fork\_cleanup\_mm:

if (p->mm)

mmput(p->mm);

bad\_fork\_cleanup\_signal:

exit\_signal(p);

bad\_fork\_cleanup\_sighand:

exit\_sighand(p);

bad\_fork\_cleanup\_fs:

exit\_fs(p); /\* blocking \*/

bad\_fork\_cleanup\_files:

exit\_files(p); /\* blocking \*/

bad\_fork\_cleanup\_semundo:

exit\_sem(p);

bad\_fork\_cleanup\_audit:

audit\_free(p);

bad\_fork\_cleanup\_security:

security\_task\_free(p);

bad\_fork\_cleanup\_policy:

#ifdef CONFIG\_NUMA

mpol\_free(p->mempolicy);

#endif

bad\_fork\_cleanup:

if (p->binfmt)

module\_put(p->binfmt->module);

bad\_fork\_cleanup\_put\_domain:

module\_put(p->thread\_info->exec\_domain->module);

bad\_fork\_cleanup\_count:

put\_group\_info(p->group\_info);

atomic\_dec(&p->user->processes);

free\_uid(p->user);

bad\_fork\_free:

free\_task(p);

goto fork\_out;

}

**2.4.2 内核线程**

内核线程是直接由内核本身启动的进程，也经常称之为守护进程。主要用于执

行下列任务：

1. 周期性地将修改的内存页与页来源块设备同步；
2. 如果内存页很少使用，则写入交换区；
3. 管理延时操作；
4. 实现文件系统的事物日志。

**分类:**

1. 线程启动后一直等待，直至内核请求线程执行某一特定操作。
2. 线程启动后周期性间隔运行检测特定资源的使用，在资源用量超出或

低于预置的限制时采取行动。

**内核线程原型：**



**注意：**1）内核线程只在CPU的管态执行，而不是用户态。

2）它们只可以访问虚拟地址空间的内核部分，但不能访问用户空间。

**惰性TLB处理**

假如内核线程之后运行的进程和之前是同一个进程，在这种情况下，

内核不需要修改用户空间地址表，地址转换后备缓冲器信息依然有效，在内核线程之后执行的进程是与此前不同的用户层进程时，才进行切换。

**内核的实现方法：**

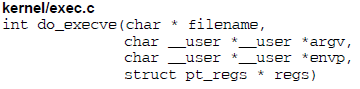
1. 将一个函数直接传递给kernei\_thread。该函数接下来负责帮助内核调用daemonize以转换为守护进程。接下来该函数首先从内核线程释放其父进程的所有资源，daemonize阻塞信号的接收，最后将init用作守护进程的父进程。创建内核线程的辅助函数：kthread\_create。
2. 使用备选方案宏kthread\_run，它会调用kthread\_create创建新线程，并立即唤醒它，还可使用kthread\_create\_cpu代替kthread\_create创建内核线程，使之绑定特定CPU。

**2.4.3 启动新程序**

Linux采用exec实现新程序的启动。

**1.execve的实现**

该系统调用的入口点是sys\_execve函数，然后委托给do\_execve函数。其原型如下:



do\_execve的代码实现流程如下图2.4.3所示。

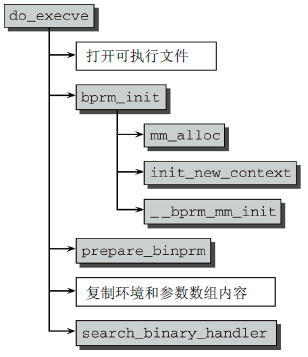


图2.4.3 do\_execve的代码流程图

sys\_execve的实现如下所示：

**asmlinkage int sys\_execve(struct pt\_regs regs)  
{  
    int  error;  
    char \* filename;  
    filename = getname((char \_\_user \*) regs.ebx);  /\*将**可执行文件的名称**装入到一个新分配的页面中 \*/  
    error = PTR\_ERR(filename);  
    if (IS\_ERR(filename))  
        goto out;  
    //执行可执行文件  
    error = do\_execve(filename,  
            (char \_\_user \* \_\_user \*) regs.ecx,  
            (char \_\_user \* \_\_user \*) regs.edx,  
            &regs);  
    if (error == 0) {  
        task\_lock(current);  
        current->ptrace &= ~PT\_DTRACE;  
        task\_unlock(current);  
          
        set\_thread\_flag(TIF\_IRET);  
    }  
    putname(filename);  
out:  
    return error;  
}**

上述系统调用do\_execve的参数在include/asm-i386/ptrace.h中被定义。

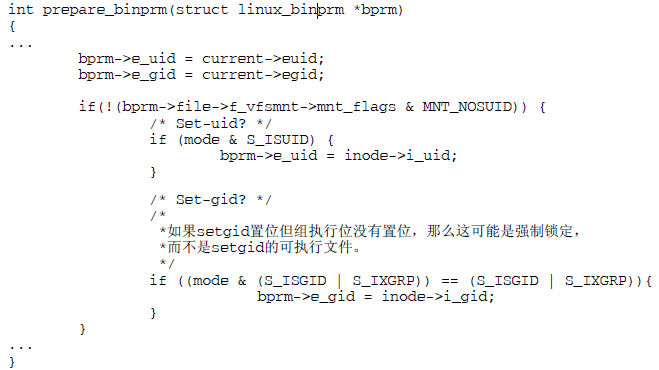
## struct pt\_regs {     long ebx;     long ecx;     long edx;     long esi;     long edi;     long ebp;     long eax;     int xds;     int xes;     long orig\_eax;     long eip;     int xcs;     long eflags;     long esp;     int xss; };

do\_execve()的实现如下所示：

**int do\_execve(char \* filename, char \_\_user \*\_\_user \*argv,  
        char \_\_user \*\_\_user \*envp,     struct pt\_regs \* regs)  
{  
    struct linux\_binprm \*bprm;  /\* 保存要执行的文件相关的数据 \*/  
    struct file \*file;  
    int retval;  
    int i;  
    retval = -ENOMEM;  
    bprm = kzalloc(sizeof(\*bprm), GFP\_KERNEL);  
    if (!bprm)  
        goto out\_ret;  
    file = open\_exec(filename); /\* 打开要执行的文件，并检查其有效性 \*/  
    retval = PTR\_ERR(file);  
    if (IS\_ERR(file))  
        goto out\_kfree;  
    /\* 在多处理器系统中才执行，用以分配负载最低的CPU来执行新程序  
    该函数在include/linux/sched.h文件中被定义如下：  
    #ifdef CONFIG\_SMP  
   extern void sched\_exec(void);  
  #else  
   #define sched\_exec() {}  
  #endif  
   sched\_exec(); \*/  
    /\* 填充linux\_binprm结构 \*/  
    bprm->p = PAGE\_SIZE\*MAX\_ARG\_PAGES-sizeof(void \*);  
    bprm->file = file;  
    bprm->filename = filename;  
    bprm->interp = filename;  
    bprm->mm = mm\_alloc();  
    retval = -ENOMEM;  
    if (!bprm->mm)  
        goto out\_file;  
    /\* 检查当前进程是否在使用LDT，如果是则给新进程分配一个LDT \*/  
    retval = init\_new\_context(current, bprm->mm);  
    if (retval  0)  
        goto out\_mm;  
    /\* 继续填充linux\_binprm结构 \*/  
    bprm->argc = count(argv, bprm->p / sizeof(void \*));  
    if ((retval = bprm->argc)  0)  
        goto out\_mm;  
    bprm->envc = count(envp, bprm->p / sizeof(void \*));  
    if ((retval = bprm->envc)  0)  
        goto out\_mm;  
    retval = security\_bprm\_alloc(bprm);  
    if (retval)  
        goto out;  
    /\* 检查文件是否可以被执行，填充linux\_binprm结构中的e\_uid和e\_gid项  
   使用可执行文件的前128个字节来填充linux\_binprm结构中的buf项 \*/  
    retval = prepare\_binprm(bprm);  
    if (retval  0)  
        goto out;  
    /\*　将文件名、环境变量和命令行参数拷贝到新分配的页面中 \*/  
    retval = copy\_strings\_kernel(1, &bprm->filename, bprm);  
    if (retval  0)  
        goto out;  
    bprm->exec = bprm->p;  
    retval = copy\_strings(bprm->envc, envp, bprm);  
    if (retval  0)  
        goto out;  
    retval = copy\_strings(bprm->argc, argv, bprm);  
    if (retval  0)  
        goto out;  
    //查询能够处理该可执行文件格式的处理函数，并调用相应的load\_library方法进行处理  
    retval = search\_binary\_handler(bprm,regs);  
    if (retval >= 0) {  
        free\_arg\_pages(bprm);  
        //执行成功  
        security\_bprm\_free(bprm);  
        acct\_update\_integrals(current);  
        kfree(bprm);  
        return retval;  
    }  
out:  
    //发生错误，返回inode，并释放资源  
    for (i = 0 ; i  MAX\_ARG\_PAGES ; i++) {  
        struct page \* page = bprm->page;  
        if (page)  
            \_\_free\_page(page);  
    }  
    if (bprm->security)  
        security\_bprm\_free(bprm);  
out\_mm:  
    if (bprm->mm)  
        mmdrop(bprm->mm);  
out\_file:  
    if (bprm->file) {  
        allow\_write\_access(bprm->file);  
        fput(bprm->file);  
    }  
out\_kfree:  
    kfree(bprm);  
out\_ret:  
    return retval;  
}**

**在上述对新进程的参数填充时,prepare\_binprm用于提供一些进程相关的的值如UID和GID。**

**同时其也维护了对SUID和SGID的简单处理：**



在确认文件文件来源卷装载时没有置位MNT\_NOSUID之后，内核会检测SUID或SGID是否置位。若SUID置位，则有效UID与inode相同；SGID和SUID类似。

Linux支持可执行文件的不同格式，标准格式是ELF，但是不同体系架构会使用许多不同的二进制格式，因此二进制格式程序不一定能够在多个架构上运行，二进制格式其实只是表示了可执行文件和内存中程序数据段、代码段等组织情况。

search\_binary\_handler用于在do\_execve结束时查找适当二进制格式，用于所要执行的特定文件，其通常根据文件起始的魔数来识别，二进制格式处理程序则将新程序加载到旧的地址空间中。

二进制格式处理程序执行的操作：

1. 释放原进程使用的所有资源；
2. 将应用程序映射到虚拟地址空间。并对下列段进行处理：

a.text包含程序的可执行代码段，start\_code和end\_code指定该段地址空间中驻留区域；

b.预先初始化的数据位于start\_data和end\_data，并映射成可执行的对应段；

c.堆用于动态内存分配，边界为start\_brk和brk；

d栈由start\_stack定义位置;

e.程序参数和环境变量也要映射如虚拟地址空间。

1. 设置进程的指令指针和寄存器。

Linux下程序在内存中的段分布如下图2.4.3所示

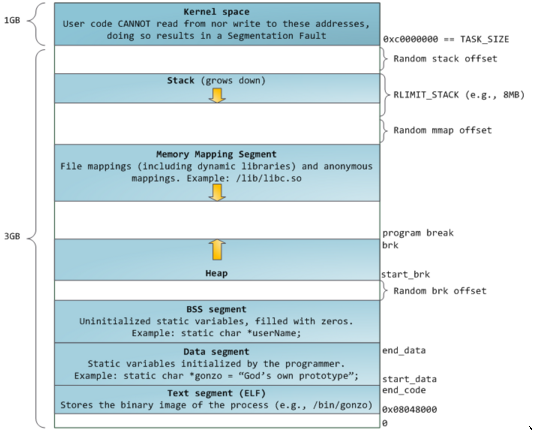
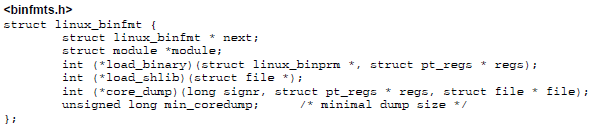


图2.4.3 Linux下程序在内存中的段分布

1. **二进制格式的解释**

在Linux内核中任何二进制格式都可用下面的结构体实例表示。



其中

1）load\_binary用于加载普通程序；

2）oad\_shlib:用于加载共享库

3）core\_dump:用于在程序错误情况下输出内存转储，随后可进行分析，以解决问题。

每种二进制格式必须使用register\_binfmt向内核注册，才能加载。

1. //linux二进制程序的参数结构体
2. **struct** linux\_binprm{
3. **char** buf[128];                        //128位的缓存
4. unsigned **long** page[MAX\_ARG\_PAGES];    //页数组
5. unsigned **long** p;                      //
6. **int** sh\_bang;                          //sh
7. **struct** inode \* inode;                 //程序点机构体指针
8. **int** e\_uid, e\_gid;                     //程序的uid,gid
9. **int** argc, envc;                       //程序的输入参数数和环境参数数
10. **char** \* filename;     /\* Name of binary \*/
11. }

2.4.4 退出进程

进程必须用exit终止，使得内核可回收进程资源。

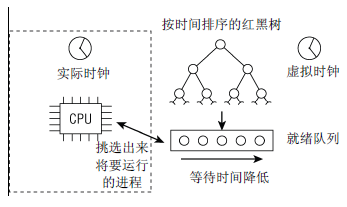
2.5 调度器的实现

调度器的任务就是使程序之间共享CPU时间，创造并行执行的假象。其可分为两个方面：一是调度策略；二是上下文切换。

**2.5.1 概观**

一般原理：按所能分配的计算能力，向系统中每个进程提供最大的公正性。

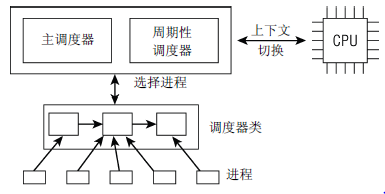
调度器对于进程等待时间的记录如图2.5.1所示。所有的可运行程序都按时间在红黑树中排序。就绪队列装备了虚拟时钟，其精确速度依赖于当前等待调度器挑选的进程的数目，约为实时时钟速度的1/4



**2.5.1 调度器记录进程等待时间**

**2.5.2 数据结构**

调度器子系统各组件概观如下图2.5.2所示。

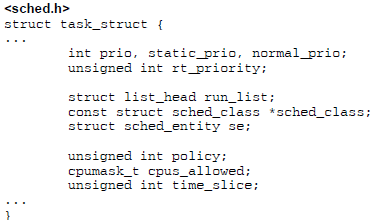


**图2.5.2 调度器子系统各组件概观**

**1．激活调度方法：**

1. 直接由进程放弃CPU;
2. 周期性机制，以固定频率运行，不时检测是否进行进程切换。

**task\_struct中与调度相关**的**成员：**



prio、normal\_prio：进程动态优先级

static\_prio:进程静态优先级，在进程启动时就被分配，可通过nice和sched\_setscheduler函数进行修改。

rt\_priority：表示实时进程优先级。最高优先级99，最低优先级0。

sched\_class:表示该进程所属的调度类。

policy:保存进程调度策略，Linux可取值为：SCHED\_NORMAL用于普通进程，SCHED\_BATCH用于非交互、CPU使用密集的批处理进程，SCHED\_IDLE权重相对较小，SCHED\_FIFO使用先进先出机制，SCHED\_RR使用循环方法，两个都用于软实时进程。

cpus\_allow：位域，用来限制执行进程的CPU。

run\_list：用于维护包含各进程的一个运行表

time\_slice:：指定进程可使用CPU的剩余时间段。

**2．调度器类**

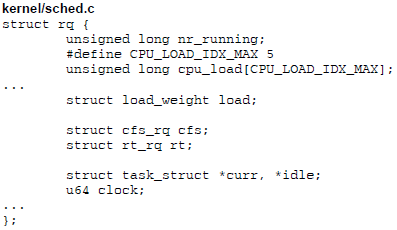
其结构如下：

1. **struct** sched\_class {
2. **const** **struct** sched\_class \*next;
3. /\* 向就绪队列添加新进程\*/
4. **void** (\*enqueue\_task) (**struct** rq \*rq, **struct** task\_struct \*p, **int** flags);   /\* 从就绪队列中删除一个进程\*/
5. **void** (\*dequeue\_task) (**struct** rq \*rq, **struct** task\_struct \*p, **int** flags);   /\* 进程自愿放弃对处理器的控制权时调用sched\_yield->yield\_task \*/
6. **void** (\*yield\_task) (**struct** rq \*rq);
7. /\* 用一个新唤醒的进程类抢占当前进程 \*/
8. **void** (\*check\_preempt\_curr) (**struct** rq \*rq, **struct** task\_struct \*p, **int** flags);
9. /\* 选择下一个要运行的进程\*/
10. **struct** task\_struct \* (\*pick\_next\_task) (**struct** rq \*rq);
11. /\* 用一个进程替代当前运行的进程 \*/
12. **void** (\*put\_prev\_task) (**struct** rq \*rq, **struct** task\_struct \*p);
14. #ifdef CONFIG\_SMP   /\* 选择就绪队列\*/
15. **int**  (\*select\_task\_rq)(**struct** rq \*rq, **struct** task\_struct \*p,
16. **int** sd\_flag, **int** flags);
18. **void** (\*pre\_schedule) (**struct** rq \*this\_rq, **struct** task\_struct \*task);
19. **void** (\*post\_schedule) (**struct** rq \*this\_rq);
20. **void** (\*task\_waking) (**struct** rq \*this\_rq, **struct** task\_struct \*task);
21. **void** (\*task\_woken) (**struct** rq \*this\_rq, **struct** task\_struct \*task);
23. **void** (\*set\_cpus\_allowed)(**struct** task\_struct \*p,
24. **const** **struct** cpumask \*newmask);
26. **void** (\*rq\_online)(**struct** rq \*rq);
27. **void** (\*rq\_offline)(**struct** rq \*rq);
28. #endif
29. /\* 设置当前执行的进程 \*/
30. **void** (\*set\_curr\_task) (**struct** rq \*rq);
31. /\* 在每次激活周期性调度器时，由周期性调度器调用 \*/
32. **void** (\*task\_tick) (**struct** rq \*rq, **struct** task\_struct \*p, **int** queued);
33. /\* 创建一个新进程\*/
34. **void** (\*task\_fork) (**struct** task\_struct \*p);
36. **void** (\*switched\_from) (**struct** rq \*this\_rq, **struct** task\_struct \*task,
37. **int** running);
38. **void** (\*switched\_to) (**struct** rq \*this\_rq, **struct** task\_struct \*task,
39. **int** running);
40. **void** (\*prio\_changed) (**struct** rq \*this\_rq, **struct** task\_struct \*task,
41. **int** oldprio, **int** running);
43. unsigned **int** (\*get\_rr\_interval) (**struct** rq \*rq,
44. **struct** task\_struct \*task);
46. #ifdef CONFIG\_FAIR\_GROUP\_SCHED
47. **void** (\*task\_move\_group) (**struct** task\_struct \*p, **int** on\_rq);
48. #endif
49. };
50. /\* 用于负载均衡\* /
51. **struct** load\_weight {
52. unsigned **long** weight, inv\_weight;
53. };

**3．就绪队列**

核心调度器用于管理活动进程的主要数据结构称之为就绪队列。每个CPU都有自己的就绪队列，一个进程不可能出现在多个就绪队列，即不能多个CPU不能运行一个进程。

就绪队列的结构如下所示。



nr\_running：制定了队列上可运行进程的数目。

load：提供了就绪队列当前负荷的度量。

cpu\_load：跟踪当前的负荷状态。

cfs和rt：嵌入的子就绪队列，分别用于完全公平调度器和实时调度器。

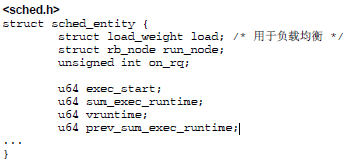
curr：指向当前运行的进程实例。

idle：指向idle进程的task\_struct实例。

clock和pre\_raw\_clock：用于实现就绪队列自身的时钟。

**4．调度实体**

调度器的操作实体结构如下：



load：指定了权重，决定了各个实体占队列总负荷的比例。

run\_node：标准的树结点，使得实体可以在红黑树上排序。

on\_rq：表示该实体当前是否在就绪队列上接受调度。

exec\_start：每次调用时，会计算当前时间和exec\_start之间的差值，exec则更新到当前时间，差值则被加到sun\_exec\_time上。

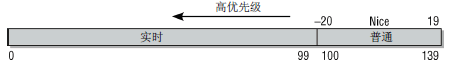
vruntime：统计在进程执行期间虚拟时钟上流逝的时间数量。

pre\_sun\_exec\_runtime：保存进程被撤销时的值。

**2.5.3 处理优先级**

**1.优先级的内核表示**

**进程的优先级的nice值在-20~+19之间。0~139表示内部优先级，0~99的范围专供实时进程使用。nice值在[-20,+19]映射到100~139之间。如图2.5.3所示。**



**图2.5.3 内核优先级标度**

**表示进程优先级宏在不同形式间转换：**

**/\* 从用户优先级的nice值和静态优先级的转换 \*/**

**#define NICE\_TO\_PRIO(nice) (MAX\_RT\_PRIO + (nice) + 20)**

**#define PRIO\_TO\_NICE(prio) ((prio) - MAX\_RT\_PRIO - 20)**

**#define TASK\_NICE(p) PRIO\_TO\_NICE((p)->static\_prio)**

**/\* 用户优先级根据调度参数转变到最佳共作级别 \*/**

**#define USER\_PRIO(p) ((p)-MAX\_RT\_PRIO)**

**#define TASK\_USER\_PRIO(p) USER\_PRIO((p)->static\_prio)**

**#define MAX\_USER\_PRIO (USER\_PRIO(MAX\_PRIO))**

**2．优先级的计算**

**static是计算的起点，假定已经设置成功。**

**则计算当前进程优先级，p->prio = effective\_prio(p);**

**static int effective\_prio(struct task\_struct \*p)**

**{**

**p->normal\_prio = normal\_prio(p);**

**/\***

**\* 如果是实时进程或已经提高到实时优先级，则保持优先级不变。**

**\* 否则返回普通优先级**

**\*/**

**if (!rt\_prio(p->prio))**

**return p->normal\_prio;**

**return p->prio;**

**}**

**普通进程优先级计算的实现：**

**static inline int normal\_prio(struct task\_struct \*p)**

**{**

**int prio;**

**if (task\_has\_rt\_policy(p))**

**prio = MAX\_RT\_PRIO-1 - p->rt\_priority;**

**else**

**prio = \_\_normal\_prio(p);**

**return prio;**

**}**

**普通进程的优先级只是返回了静态优先级而已。**

**static inline int \_\_normal\_prio(struct task\_struct \*p)**

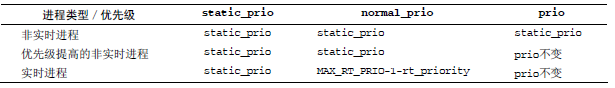
**{**

**return p->static\_prio;**

**}**

**下面表格对上述的实现做了总结，如表2.5.3所示**

**表2.5.3 对各种类型的进程计算优先级**



**注意：在进程支出子进程时，子进程的静态优先级继承自父进程。子进程的动态优先级则设置为父进程的普通父进程，这确保了实时互斥量引起的优先级提高不会传递给子进程。**

**3．计算负荷权重**

**进程的重要性由优先级和task\_struct->se.load的负荷权重。set\_load\_weight负责根据进程类型及其静态优先级计算负荷权重。**

**其实现过程如下:**

**static void set\_load\_weight(struct task\_struct \*p)**

**{**

**if (task\_has\_rt\_policy(p)) {**

**p->se.load.weight = prio\_to\_weight[0] \* 2;**

**p->se.load.inv\_weight = prio\_to\_wmult[0] >> 1;**

**return;**

**}**

**/\***

**\* SCHED\_IDLE tasks get minimal weight:**

**\*/**

**if (p->policy == SCHED\_IDLE) {**

**p->se.load.weight = WEIGHT\_IDLEPRIO;**

**p->se.load.inv\_weight = WMULT\_IDLEPRIO;**

**return;**

**}**

**p->se.load.weight = prio\_to\_weight[p->static\_prio - MAX\_RT\_PRIO];**

**p->se.load.inv\_weight = prio\_to\_wmult[p->static\_prio - MAX\_RT\_PRIO];**

**}**

**负荷的权重包含在load\_weight中：**



**一般进程每降低一个nice值，则看获得10%的CPU时间。内核将优先级转换为权重值，下面是这个转换表。**

**static const int prio\_to\_weight[40] = {**

**/\* -20 \*/ 88761, 71755, 56483, 46273, 36291,**

**/\* -15 \*/ 29154, 23254, 18705, 14949, 11916,**

**/\* -10 \*/ 9548, 7620, 6100, 4904, 3906,**

**/\* -5 \*/ 3121, 2501, 1991, 1586, 1277,**

**/\* 0 \*/ 1024, 820, 655, 526, 423,**

**/\* 5 \*/ 335, 272, 215, 172, 137,**

**/\* 10 \*/ 110, 87, 70, 56, 45,**

**/\* 15 \*/ 36, 29, 23, 18, 15,**

**};**

**不仅进程，就绪队列也关联到了一个负载权重。在进程被加载进就绪队列时，内核会调用inc\_nr\_running，还将进程的权重添加到就绪队列的权重中。**

**static inline void update\_load\_add(struct load\_weight \*lw, unsigned long inc)**

**{**

**lw->weight += inc;**

**}**

**static inline void update\_load\_sub(struct load\_weight \*lw, unsigned long dec)**

**{**

**lw->weight -= dec;**

**}**

**static void inc\_nr\_running(struct task\_struct \*p, struct rq \*rq)**

**{**

**rq->nr\_running++;**

**inc\_load(rq, p);**

**}**

**2.5.4 核心调度器**

**1．周期性调度器**

**周期性调度器的主要任务：**

**1）管理内核中与整个系统和各个进程的调度相关的统计量，其主要操作是各种计数器加1。**

**2）激活负责当前进程的调度类的周期性调度方法。**

**其具体实现如下：**

**void scheduler\_tick(void)**

**{**

**int cpu = smp\_processor\_id();**

**struct rq \*rq = cpu\_rq(cpu);**

**struct task\_struct \*curr = rq->curr;**

**u64 next\_tick = rq->tick\_timestamp + TICK\_NSEC;**

**spin\_lock(&rq->lock);**

**\_\_update\_rq\_clock(rq);**

**/\***

**\* Let rq->clock advance by at least TICK\_NSEC:**

**\*/**

**if (unlikely(rq->clock < next\_tick))**

**rq->clock = next\_tick;**

**rq->tick\_timestamp = rq->clock;**

**update\_cpu\_load(rq);**

**if (curr != rq->idle) /\* FIXME: needed? \*/**

**curr->sched\_class->task\_tick(rq, curr);**

**spin\_unlock(&rq->lock);**

**#ifdef CONFIG\_SMP**

**rq->idle\_at\_tick = idle\_cpu(cpu);**

**trigger\_load\_balance(rq, cpu);**

**#endif**

**}**

**2．主调度器**

**主调度器实现：**

**asmlinkage void \_\_sched schedule(void)**

**{**

**struct task\_struct \*prev, \*next;**

**long \*switch\_count;**

**struct rq \*rq;**

**int cpu;**

**need\_resched:**

**preempt\_disable();**

**cpu = smp\_processor\_id();**

**rq = cpu\_rq(cpu);**

**rcu\_qsctr\_inc(cpu);**

**prev = rq->curr;**

**switch\_count = &prev->nivcsw;**

**release\_kernel\_lock(prev);**

**need\_resched\_nonpreemptible:**

**schedule\_debug(prev);**

**/\***

**\*更新就绪队列的时钟，并清除当前运行进程task\_struct中的重调度标志**

**TIF\_NEED\_RESCHED。**

**\*/**

**local\_irq\_disable();**

**\_\_update\_rq\_clock(rq);**

**spin\_lock(&rq->lock);**

**clear\_tsk\_need\_resched(prev);**

**if (prev->state && !(preempt\_count() & PREEMPT\_ACTIVE)) {**

**if (unlikely((prev->state & TASK\_INTERRUPTIBLE) &&**

**unlikely(signal\_pending(prev)))) {**

**prev->state = TASK\_RUNNING;**

**} else {**

**deactivate\_task(rq, prev, 1);**

**}**

**switch\_count = &prev->nvcsw;**

**}**

**if (unlikely(!rq->nr\_running))**

**idle\_balance(cpu, rq);**

**prev->sched\_class->put\_prev\_task(rq, prev);**

**next = pick\_next\_task(rq, prev);**

**sched\_info\_switch(prev, next);**

**if (likely(prev != next)) {**

**rq->nr\_switches++;**

**rq->curr = next;**

**++\*switch\_count;**

**context\_switch(rq, prev, next); /\* unlocks the rq \*/**

**} else**

**spin\_unlock\_irq(&rq->lock);**

**if (unlikely(reacquire\_kernel\_lock(current) < 0)) {**

**cpu = smp\_processor\_id();**

**rq = cpu\_rq(cpu);**

**goto need\_resched\_nonpreemptible;**

**}**

**preempt\_enable\_no\_resched();**

**if (unlikely(test\_thread\_flag(TIF\_NEED\_RESCHED)))**

**goto need\_resched;**

**}**

1. **与fork的交互**

**每当fork()或其他变体创建新进程时，调度器都可能用sched\_fork()函数挂钩到该进程。若其在单处理器上，其实质上做了三个操作：初始化新进程与调度相关字段、建立数据结构、确定动态优先级。**

**源码实现：**

**void sched\_fork(struct task\_struct \*p, int clone\_flags)**

**{**

**int cpu = get\_cpu();**

**/\*初始化数据结构\*/**

**\_\_sched\_fork(p);**

**#ifdef CONFIG\_SMP**

**/\* \*/**

**cpu = sched\_balance\_self(cpu, SD\_BALANCE\_FORK);**

**#endif**

**set\_task\_cpu(p, cpu);**

**/\* 确认没有将提高的优先级泄漏到子进程 \*/**

**p->prio = current->normal\_prio;**

**if (!rt\_prio(p->prio))/\* ������̲���ʵʱ���ȼ��������ǽ���CFS���� \*/**

**p->sched\_class = &fair\_sched\_class;**

**#if defined(CONFIG\_SCHEDSTATS) || defined(CONFIG\_TASK\_DELAY\_ACCT)**

**if (likely(sched\_info\_on()))**

**memset(&p->sched\_info, 0, sizeof(p->sched\_info));**

**#endif**

**#if defined(CONFIG\_SMP) && defined(\_\_ARCH\_WANT\_UNLOCKED\_CTXSW)**

**p->oncpu = 0;**

**#endif**

**#ifdef CONFIG\_PREEMPT**

**/\* 开启内核抢占 \*/**

**task\_thread\_info(p)->preempt\_count = 1;**

**#endif**

**put\_cpu();**

**}**

1. **上下文切换**

**上下文切换：内核选择新进程后，处理与多任务相关的技术细节。**

**上下文切换主要通过两个函数完成：switch\_mm()和****switch\_to()。**

**switch\_mm：更换通过task\_struct->mm描述的内存管理上下文。**

**switch\_to:切换处理器寄存器内容和内核栈。**

**源码实现：**

**static inline void**

**context\_switch(struct rq \*rq, struct task\_struct \*prev,**

**struct task\_struct \*next)**

**{**

**struct mm\_struct \*mm, \*oldmm;**

**/\* 执行特定体系结构的任务切换准备工作，一般不需要做特定的任务 \*/**

**prepare\_task\_switch(rq, prev, next);**

**mm = next->mm;**

**oldmm = prev->active\_mm;**

**arch\_enter\_lazy\_cpu\_mode();**

**if (unlikely(!mm)) {/\* 新进程是内核进程，不需要切换内存管理上下文 \*/**

**next->active\_mm = oldmm;**

**atomic\_inc(&oldmm->mm\_count);**

**/\* 通知底层结构不需要切换用户态虚拟地址空间。也不需要立即刷新tlb \*/**

**enter\_lazy\_tlb(oldmm, next);**

**} else/\* 需要切换内存上下文 \*/**

**switch\_mm(oldmm, mm, next);**

**if (unlikely(!prev->mm)) {/\* 进程是内核进程 \*/**

**/\* 断开与借用地址空间的联系 \*/**

**prev->active\_mm = NULL;**

**rq->prev\_mm = oldmm;**

**}**

**#ifndef \_\_ARCH\_WANT\_UNLOCKED\_CTXSW**

**spin\_release(&rq->lock.dep\_map, 1, \_THIS\_IP\_);**

**#endif**

**/\* 执行体系结构相关的寄存器内容和内核栈切换 \*/**

**switch\_to(prev, next, prev);**

**/\* 编译屏障 \*/**

**barrier();**

**/\* 进行一些清理工作，释放一些锁 \*/**

**finish\_task\_switch(this\_rq(), prev);**

**}**

**2.6 完全公平调度类**

**完全公平调度器的信息**

**static const struct sched\_class fair\_sched\_class = {**

**.next = &idle\_sched\_class,**

**.enqueue\_task = enqueue\_task\_fair,**

**.dequeue\_task = dequeue\_task\_fair,**

**.yield\_task = yield\_task\_fair,**

**.check\_preempt\_curr = check\_preempt\_wakeup,**

**.pick\_next\_task = pick\_next\_task\_fair,**

**.put\_prev\_task = put\_prev\_task\_fair,**

**#ifdef CONFIG\_SMP**

**.load\_balance = load\_balance\_fair,**

**.move\_one\_task = move\_one\_task\_fair,**

**#endif**

**.set\_curr\_task = set\_curr\_task\_fair,**

**.task\_tick = task\_tick\_fair,**

**.task\_new = task\_new\_fair,**

**};**

**2.6.1 CFS就绪队列的数据结构**

**struct cfs\_rq {**

**struct load\_weight load; /\*维护这些进程的累积负荷值\*/**

**unsigned long nr\_running; /\*计算队列上可运行进程的数目\*/**

**u64 exec\_clock;**

**u64 min\_vruntime; /\* 跟踪记录队列上所有进程的最小虚拟运行时间\*/**

**struct rb\_root tasks\_timeline; /\*用于在按时间排序的红黑树中管理进程\*/**

**struct rb\_node \*rb\_leftmost; //指向最需要被调度的进程**

**struct rb\_node \*rb\_load\_balance\_curr; //红黑树中用于负载平衡的进程**

**/\* 'curr' points to currently running entity on this cfs\_rq.**

**\* It is set to NULL otherwise (i.e when none are currently running).**

**\*/**

**struct sched\_entity \*curr; //指向当前执行进程的可调度实体**

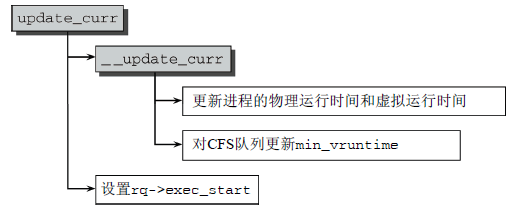
**unsigned long nr\_spread\_over;**

**};**

**2.6.2 CFS操作**

**1、虚拟时钟**

**完全公平调度算法依赖于虚拟时钟，用以度量等待进程在完全公平系统中所能得到的CPU时间，虚拟时钟是根据实际时钟与每个进程相关的负荷权重推算出来的，其计算都在update\_curr中被执行。update\_curr代码流程图如图2.6.2所示，后面并附实现源码。**



**图2.6.2 update\_curr代码流程图**

**源码：**

**static void update\_curr(struct cfs\_rq \*cfs\_rq)**

**{**

**struct sched\_entity \*curr = cfs\_rq->curr; //确定就绪队列的当前执行进程**

**u64 now = rq\_of(cfs\_rq)->clock; //获取主调度器就绪队列的实际时钟值**

**unsigned long delta\_exec;**

**if (unlikely(!curr)) //如果就绪对列上当前没有进程正在执行，则无事可做。**

**return;**

**/\*计算当前和上一次更新负荷统计量时两次的时间差**

**\*/**

**delta\_exec = (unsigned long)(now - curr->exec\_start);**

**\_\_update\_curr(cfs\_rq, curr, delta\_exec);//将剩余工作转交给\_update\_curr。**

**curr->exec\_start = now;**

**if (entity\_is\_task(curr)) {**

**struct task\_struct \*curtask = task\_of(curr);**

**cpuacct\_charge(curtask, delta\_exec);**

**}**

**}**

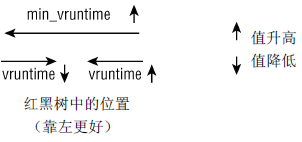
**完全公平调度器的真正关键点：红黑树的排序过程**

**1）键值较小的结点，排序位置越靠左，因此就会更快被调度。在进程运行**

**时，其vruntime稳定地增加，它在红黑树中总是向右移动的。越重要的今后才呢过vruntime增加越慢，因此它们向右移动速度越慢。**

**2）如果进程进入睡眠，则vruntime保持不变，因为每个队列min\_vruntime同时会增加，那么睡眠进程醒来，在红黑树位置更靠左。**

**如下图2.6.2.1所示。**



**图2.6.2.1 每个调度实体和每个队列的虚拟时间对进程在红黑树中为位置的影响**

**源码实现：**

**static inline u64 max\_vruntime(u64 min\_vruntime, u64 vruntime)**

**{**

**s64 delta = (s64)(vruntime - min\_vruntime);**

**if (delta > 0)**

**min\_vruntime = vruntime;**

**return min\_vruntime;**

**}**

**static inline u64 min\_vruntime(u64 min\_vruntime, u64 vruntime)**

**{**

**s64 delta = (s64)(vruntime - min\_vruntime);**

**if (delta < 0)**

**min\_vruntime = vruntime;**

**return min\_vruntime;**

**}**

**2、延迟跟踪**

**良好的调度延迟：保证每个可运行的进程都应该至少运行一次的某个时间间隔。默认值为20 000 000ns或20ms。**

**unsigned int sysctl\_sched\_latency = 20000000ULL;**

**控制在一个延迟周期中处理最大活动进程数目:sched\_nr\_latency**

**static unsigned int sched\_nr\_latency = 5;**

**某个可调度实体表示的给定进程分配到的时间：**

**static u64 sched\_slice(struct cfs\_rq \*cfs\_rq, struct sched\_entity \*se)**

**{**

**u64 slice = \_\_sched\_period(cfs\_rq->nr\_running);**

**slice \*= se->load.weight;**

**do\_div(slice, cfs\_rq->load.weight);**

**return slice;**

**}**

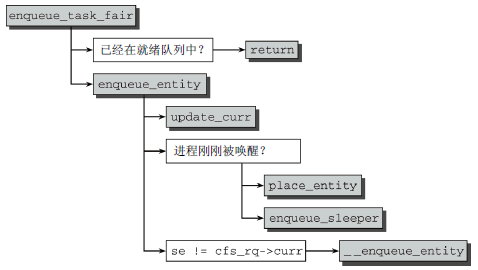
**就绪队列的各个权重是队列上左右活动负荷权重的累加和，但是时间段是按**

**实际时间给出的。**

**2.6.3 队列操作**

**操作就绪队列的成员：enqueue\_task\_fair和dequeue\_task\_fair。**

**向就绪队列放置新进程 ：**



**图2.6.3 enqueue\_task\_fair的代码流程图**

**源码实现：**

**/\* 若该指定入队进程最近被唤醒为运行状态，此时wakeup=1；若此前就是可运行的，则wakeup=0\*/**

**static void enqueue\_task\_fair(struct rq \*rq, struct task\_struct \*p, int wakeup)**

**{**

**struct cfs\_rq \*cfs\_rq;**

**struct sched\_entity \*se = &p->se;**

**for\_each\_sched\_entity(se) {**

**if (se->on\_rq)**

**break;**

**cfs\_rq = cfs\_rq\_of(se);**

**enqueue\_entity(cfs\_rq, se, wakeup);**

**wakeup = 1;**

**}**

**}**

**若此前进程在运行，其虚拟时间仍然有效；若此前进程在睡眠，那么需要**

**place\_entity()调整进程的虚拟运行时间。**

**static void place\_entity(struct cfs\_rq \*cfs\_rq, struct sched\_entity \*se, int initial)**

**{**

**u64 vruntime;**

**vruntime = cfs\_rq->min\_vruntime;**

**if (sched\_feat(TREE\_AVG)) {**

**struct sched\_entity \*last = \_\_pick\_last\_entity(cfs\_rq);**

**if (last) {**

**vruntime += last->vruntime;**

**vruntime >>= 1;**

**}**

**} else if (sched\_feat(APPROX\_AVG) && cfs\_rq->nr\_running)**

**vruntime += sched\_vslice(cfs\_rq)/2;**

**/\***

**\* 内核已经承诺在当前的延迟周期内使所有活动进程都至少运行一次，通过减去sysctl\_sched\_latency，确保新唤醒的进程只有在当前延迟周期结束后才能运行。**

**\*/**

**if (initial && sched\_feat(START\_DEBIT))**

**vruntime += sched\_vslice\_add(cfs\_rq, se);{**

**if (!initial) {**

**/\* sleeps upto a single latency don't count. \*/**

**if (sched\_feat(NEW\_FAIR\_SLEEPERS) && entity\_is\_task(se))**

**vruntime -= sysctl\_sched\_latency;**

**/\*若se->vruntime比先前计算的差值大，则将其作为进程的vruntime，从而使得其位于进程在红黑树较左位置，更早被执行. \*/**

**vruntime = max\_vruntime(se->vruntime, vruntime);**

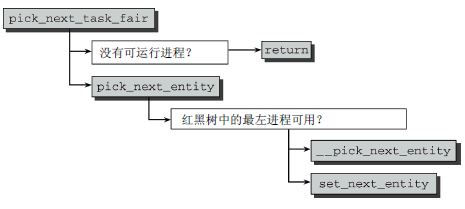
**}**

**se->vruntime = vruntime;**

**}**

**2.6.4 选择下一个进程**

**实现函数：pick\_next\_task\_fair，其代码流程图如下2.6.4。**



**图2.6.24 pick\_next\_task\_fair代码流程图**

**源码实现：**

**static struct task\_struct \*pick\_next\_task\_fair(struct rq \*rq)**

**{**

**struct cfs\_rq \*cfs\_rq = &rq->cfs;**

**struct sched\_entity \*se;**

**if (unlikely(!cfs\_rq->nr\_running)) //如果计数器为0，即当前队列上没有可运行进程，则无事可做，返回**

**return NULL;**

**do {**

**se = pick\_next\_entity(cfs\_rq); //选择下一个进程**

**cfs\_rq = group\_cfs\_rq(se); //将其加入自己组内**

**} while (cfs\_rq);**

**return task\_of(se);**

**}**

**static void set\_next\_entity(struct cfs\_rq \*cfs\_rq, struct sched\_entity \*se)**

**{**

**/\* 'current' is not kept within the tree. \*/**

**/\* 当前进程没有在树内 \*/**

**if (se->on\_rq) {**

**/\***

**\* 在得到任何CPU可执行片段前任何进程不得不被加入就绪队列**

**\* 所以它花时间在等待就绪队列**

**\*/**

**update\_stats\_wait\_end(cfs\_rq, se); //更新等待的状态**

**\_\_dequeue\_entity(cfs\_rq, se); //从树中将其移除**

**}**

**update\_stats\_curr\_start(cfs\_rq, se); //建立进程和就绪队列的联系**

**cfs\_rq->curr = se; //标记当前运行的进程**

**#ifdef CONFIG\_SCHEDSTATS**

**/\***

**\* Track our maximum slice length, if the CPU's load is at**

**\* least twice that of our own weight (i.e. dont track it**

**\* when there are only lesser-weight tasks around):**

**\*/**

**if (rq\_of(cfs\_rq)->load.weight >= 2\*se->load.weight) {**

**se->slice\_max = max(se->slice\_max,**

**se->sum\_exec\_runtime - se->prev\_sum\_exec\_runtime);**

**}**

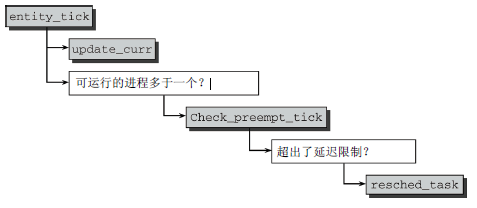
**#endif**

**se->prev\_sum\_exec\_runtime = se->sum\_exec\_runtime; //保存此前的设置**

**}**

**2.6.5 处理周期性调度器**

**形式上task\_tick\_fair\_——>实际entity\_tick。其代码流程图如下2.6.5所示。**



**图2.6.5 entity\_tick()代码流程图**

**实现源码：**

**static void entity\_tick(struct cfs\_rq \*cfs\_rq, struct sched\_entity \*curr)**

**{**

**/\* 更新当前进程的运行时间 \*/**

**update\_curr(cfs\_rq);**

**/\* 如果队列上可用进程数量不足两个，则不必进行切换 \*/**

**if (cfs\_rq->nr\_running > 1 || !sched\_feat(WAKEUP\_PREEMPT))**

**/\* 否则，处理CFS进程抢占 \*/**

**check\_preempt\_tick(cfs\_rq, curr);**

**}**

**static void**

**check\_preempt\_tick(struct cfs\_rq \*cfs\_rq, struct sched\_entity \*curr)**

**{**

**unsigned long ideal\_runtime, delta\_exec;**

**/\* 计算进程的份额 \*/**

**ideal\_runtime = sched\_slice(cfs\_rq, curr);**

**/\* 计算已经运行的时间 \*/**

**delta\_exec=curr->sum\_exec\_runtime - curr->prev\_sum\_exec\_runtime;**

**/\* 进程运行的时间比期望的份额更长 \*/**

**if (delta\_exec > ideal\_runtime)**

**/\* 发出重新调度请求 \*/**

**resched\_task(rq\_of(cfs\_rq)->curr);**

**}**

**2.6.6 唤醒抢占**

**在使用try\_to\_wake\_up和wake\_up\_new\_task唤醒进程时，内核会使用check\_preempt\_curr看是否有新进程可抢占当前运行进程。完全公平调度器处理的进程，由check\_preempt\_wakeup()进行检测。**

**若新唤醒的进程为实时进程则会立即请求重调度，会抢占CFS进程。**

**\* 当进程被try\_to\_wake\_up和wake\_up\_new\_task唤醒时，调用此函数检查是否可以抢占当前进程**

**\*/**

**static void check\_preempt\_wakeup(struct rq \*rq, struct task\_struct \*p)**

**{**

**struct task\_struct \*curr = rq->curr;**

**struct cfs\_rq \*cfs\_rq = task\_cfs\_rq(curr);**

**struct sched\_entity \*se = &curr->se, \*pse = &p->se;**

**unsigned long gran;**

**/\* 新唤醒的进程是实时进程 \*/**

**if (unlikely(rt\_prio(p->prio))) {**

**update\_rq\_clock(rq);**

**update\_curr(cfs\_rq);**

**/\* 实时进程总是抢占CFS进程 \*/**

**resched\_task(curr);**

**return;**

**}**

**/\***

**\* Batch tasks do not preempt (their preemption is driven by**

**\* the tick):**

**\*/**

**/\* 新进程是BATCH进程，它不抢占其他进程 \*/**

**if (unlikely(p->policy == SCHED\_BATCH))**

**return;**

**/\* 没有配置唤醒抢占功能 \*/**

**if (!sched\_feat(WAKEUP\_PREEMPT))**

**return;**

**while (!is\_same\_group(se, pse)) {**

**se = parent\_entity(se);**

**pse = parent\_entity(pse);**

**}**

**/\* 计算最小运行时间，默认是4ms \*/**

**gran = sysctl\_sched\_wakeup\_granularity;**

**/\* 计算最小运行时间对应的虚拟时间 \*/**

**if (unlikely(se->load.weight != NICE\_0\_LOAD))**

**gran = calc\_delta\_fair(gran, &se->load);**

**/\***

**\* 新进程的虚拟运行时间加上最小运行时间仍然小于当前进程的配额**

**\* 说明新进程优先级较高，抢占当前进程**

**\*/**

**if (pse->vruntime + gran < se->vruntime)**

**resched\_task(curr);**

**}**

**2.6.7 处理新进程**

**创建新进程后需调用挂钩函数task\_new\_fair(),其会处理子进程是否父进程之前运行。**

**源码实现：**

**static void task\_new\_fair(struct rq \*rq, struct task\_struct \*p)**

**{**

**struct cfs\_rq \*cfs\_rq = task\_cfs\_rq(p);**

**struct sched\_entity \*se = &p->se, \*curr = cfs\_rq->curr;**

**int this\_cpu = smp\_processor\_id();**

**sched\_info\_queued(p);**

**/\* 更新当前进程的运行时间 \*/**

**update\_curr(cfs\_rq);**

**/\* 将新进程放置到红黑树中，initial参数表示是新创建的进程 \*/**

**place\_entity(cfs\_rq, se, 1);**

**if (sysctl\_sched\_child\_runs\_first && this\_cpu == task\_cpu(p) &&/\* 子进程先运行，子进程与父进程在同一CPU上 \*/**

**curr && curr->vruntime < se->vruntime) {/\* 当前进程的虚拟运行时间小于子进程的运行时间 \*/**

**/\***

**\* Upon rescheduling, sched\_class::put\_prev\_task() will place**

**\* 'current' within the tree based on its new key value.**

**\*/**

**/\* 交换二者的运行时间，随后会将当前进程插入到红黑树合适的位置 \*/**

**swap(curr->vruntime, se->vruntime);**

**}**

**/\* 将新进程加入到运行队列并重新调度 \*/**

**enqueue\_task\_fair(rq, p, 0);**

**resched\_task(rq->curr);**

**}**

**2.7 实时调度类**

**实时进程其优先级比普通进程高，其static\_prio值总是低于普通农进程。**

**2.7.1 性质**

**Linux支持的两种实时调度类：**

**1）循环进程:有时间片，其值在进程运行会减少，在所有进程的时间段都到期后则该值会重置为初始值，这个确保了在优先级相同情况下它们总是依次执行。**

**2）先进先出进程：没有时间片，在被调度器选择执行后，可执行任意长时间。**

**2.7.2 数据结构**

**const struct sched\_class rt\_sched\_class = {**

**.next = &fair\_sched\_class,**

**.enqueue\_task = enqueue\_task\_rt,**

**.dequeue\_task = dequeue\_task\_rt,**

**.yield\_task = yield\_task\_rt,**

**.check\_preempt\_curr = check\_preempt\_curr\_rt,**

**.pick\_next\_task = pick\_next\_task\_rt,**

**.put\_prev\_task = put\_prev\_task\_rt,**

**#ifdef CONFIG\_SMP**

**.load\_balance = load\_balance\_rt,**

**.move\_one\_task = move\_one\_task\_rt,**

**#endif**

**.set\_curr\_task = set\_curr\_task\_rt,**

**.task\_tick = task\_tick\_rt,**

**};**

**具有相同优先级的所有实时进程都保存在一个链表中，该表头为**

**active.queue[prio],其对应与active.bitmap位图中的一个比特位。若链**

**表中有实时进程，则置位，反之则反。实时就绪队列结构图如下图2.7.2**

**所示，可以发现就绪队列实质就是一个双向链表而已。**

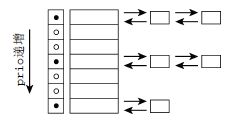


图2.7.2 实时调度器的就绪队列

注意：实时调度类中update\_cur()用sum\_exec\_runtime记录的是实际时间，不再是虚拟时间。

2.7.3 调度器操作

调度器的操作就是进程的入队和出队，在这里实现则只需用p->prio作索引访问数组queue即可。若队列中至少有一个进程，则对位图中对应比特进行值为置位；若无进程，则清除位图中对应的比特。注意：新进程总是在队末。

选择下一个将要执行的进程：pick\_next\_task\_rt(),其代码流程如下图2.7.3所示。

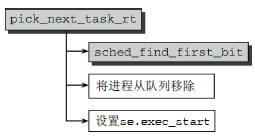


图2.7.3 将要执行的进程

首先使用sched\_find\_first\_bit找到active\_bitmap第一个置位的比特，取出所选链表的第一个进程，并将se.exec\_start置为就绪队列的当前实际时间时钟值。

若是周期调度，则其可运行任意长时间，必须使用yield系统调用显式将控制权传递另一进程。‘

源代码：

/\* 实时进程的周期性调度 \*/

static void task\_tick\_rt(struct rq \*rq, struct task\_struct \*p)

{

update\_curr\_rt(rq);

if (p->policy != SCHED\_RR)/\* 对FIFO来说，不参与周期性调度 \*/

return;

/\* 递减RR进程的运行周期，如果还有时间片，则退出 \*/

if (--p->time\_slice)

return;

/\* RR时间片用完，重新初始化为100ms \*/

p->time\_slice = DEF\_TIMESLICE;

/\* 进程不是唯一的实时进程 \*/

if (p->run\_list.prev != p->run\_list.next) {

/\* 将进程插入到列队末尾，并设置重新调度标志 \*/

requeue\_task\_rt(rq, p);

set\_tsk\_need\_resched(p);

}

}

如果是循环进程，则减少其时间片。若未超过其时间段，则继续执行；若超过其时间段，则计数器归0，其值重置为DEF\_TIMESLICE。若该进程不是链表中唯一的进程，则重新排队到末尾，通过set\_tsk\_need\_resched设置TIF\_NEED\_CHED标志，继续请求调度。

当然普通进程也可以通过sched\_setscheduler()转化为实时进程。它首先使用deactive\_task将进程从队列中移除，然后在task\_struct设置实时优先级和调度类，最后重新激活该进程。

2.8 调度器增强

2.8.1 SMP调度

为了确保多处理器的良好调度，需要考虑一下几个问题：

1）CPU负荷应尽可能公平地在所有处理器上共享；

2）进程与系统中某些处理器的亲和性必须是可设置的；

3）内核必须能够将进程在CPU间进行迁移。

1、数据结构的扩展

为了解决上述问题，在SMP系统上们每个调度器类中都必须增加两个额外 的函数：load\_balance()和move\_one\_task()。

load\_balance：允许从最忙的就绪队列分配多个进程到当前CPU，但移动的负荷不能比max\_load\_move更多。

move\_one\_task:从最忙碌的就绪队列移出一个进程，迁移到当前CPU就绪队列。

负载均衡处理过程如下图2.8.1所示。

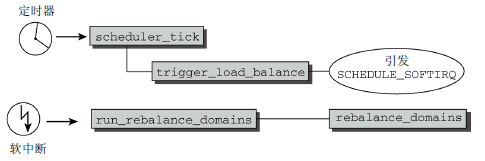
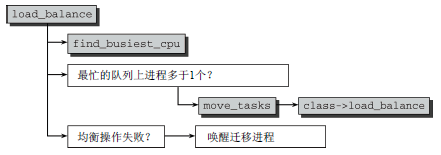


图2.8.1 负载均衡处理过程

在SMP系统中，周期性调度函数scheduler\_tick完成上述任务后，会调用trigger\_load\_balance(),然后产生软中断，以确保会在合适的时候执行run\_balance\_domains，最终实现rebalance。

load\_balance的代码实现流程如图2.8.1.1。



2.8.1.1 load\_balance的代码实现流程

注意：

1）在选择被迁移的进程时，必须保证该进程没有运行或刚结束运行。

2）能够根据其亲和性可以在当前队列关联的处理器上执行。

2. 迁移进程

目的：

1）用于完成调度器的迁移请求；

2）用于实现主动均衡。

迁移线程（migration\_thread）其代码流程如下图2.8.1.2所示。

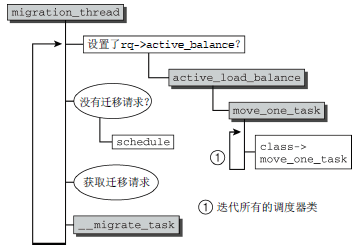


图2.8.1.2 迁移线程的代码流程

迁移线程内部是一个无限循环，无事可做时则进入睡眠状态。其首先会检测是否需要主动均衡，若需要则调用active\_load\_balance，该函数会从当前就绪队列移出一个进程至发起请求的CPU就绪队列。

完成主动负载均衡后，迁移线程会检测是否有调度器发出的待决请求，若无则重新获取；否则使用\_migrate\_task完成请求。

3. 核心调度器的改变

在SMP系统中还需要对核心调度器的一些现存方法做一些修改。

1）在使用exec新启一个进程时，是移动进程到另一个CPU的最佳时机，也不会带来负面效应。exec会调用挂钩函数sched\_exec()，该函数流程图如下图2.8.1.3所示。

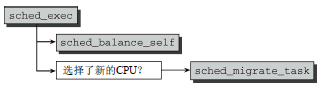


图2.8.1.3 sched\_exec()实现流程

sched\_exec()会调用sched\_balance\_self挑选当前负荷最少的CPU，若不是当前CPU，则会使用sched\_migrate\_task，向迁移线程发送一个请求。

2）完全公平调度调度的粒度和CPU数目成正比。sysctl\_sched\_min\_granularity和sysctl\_sched\_latency都乘校正因子(1+log2(nr\_cpus))，其中nr\_cpus代表CPU数目。两个其上限为200ms，sysctl\_sched\_wakeup\_granularity也要进行同样的操作，但是其无上限。

2.8.2 调度域和控制组

调度器通过可调度实体与进程进行交互，其并直接与进程进行交互。进程位于不同的组中，调度器通过首先使得这些组之间保证公平，然后在组内保持公平。

当然内核还提供了控制组技术去使得调度公平。该特性通过文件系统cgroups可创建任意的进程组集合，如2.8.2所示。

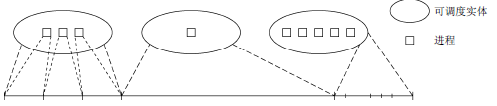


图2.8.2

1. 内存管理

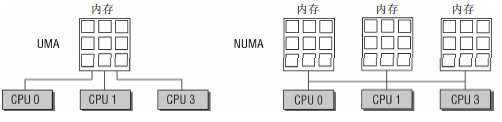
**内存管理是内核最复杂也是最重要的一部分。我们知道在访问内存时，若所用虚拟地址与内核区域的起始地址之间的偏移量不超出可用物理内存的长度，那么该虚拟地址会自动关联到物理页帧；若虚拟地址空间的内部部分小于CPU理论地址空间的最大长度，那么物理内存比可映射到内核空间的数量要多，针对这种情况我们必须借助高端内存来管理多余内存。在IA-32中，其可直接管理的内存数量最大值为896M，超过该值小于4G则通过高端内存寻址。现代IA-32则通过启用PAE模式管理里多达63G内存，PAE模式为内存指针提供了额外的比特位，故而可实现该目的。**

**3.1 两类计算机以不同方法管理物理内存**

**1）UMA计算机（一致性内存访问）将可用内存以连续方式组织起来。**

**2）NUMA计算机（非一致性内存访问）总是多处理计算机，该计算机各个CPU都有本地内存，以支持快速访问，且各处理器通过总线连接以支持CPU的本地内存访问。**

**图3.1说明了两种方法的差别。**



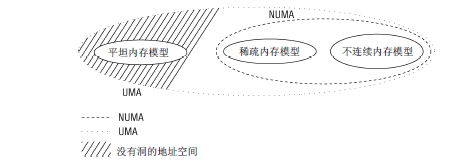
**图3.1.1 UMA和NUMA系统**

**两种类型计算机也可以混合，UMA中内存不连续，存在较大的洞，造成内存碎片化。**

**在这里内核对于内存的访问有三种配置选项：**

1. **FLATMEM：内核的默认值。**
2. **DISCONFIGMEM:性能相对更稳定，但不具备内存热插拔等新特性。**
3. **SPARSEMEM:更多的实验性，其中进行了简单性能优化**

**与内存布局相关的可能的配置选项，如图3.1.2**



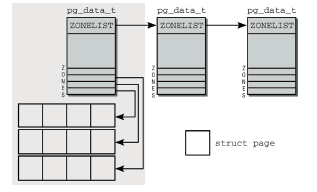
**图3.1.2 UMA和NUMA计算机可能的内存配置：平坦模型、稀疏模型、不连续模型**

**3.2 (N)UMA模型中的内存组织**

**内核对一致性和非一致性内存访问系统使用相同的数据结构，因此其针对不同的内存布局算法基本相同。**

**3.2.1 概述**

**首先内核划分结点，每个结点连接一个处理器，结点表示为pg\_data\_t。每个结点又从上到下细分为高端内存区域、通用普通内存区、内存域。其中高端内存域无法进行直接映射。如下图3.2.1所示。**



**图3.2.1**

**内核中引入了以下常量枚举系统中所有内存。**

**<mmzone.h>**

**enum zone\_type {  
#ifdef CONFIG\_ZONE\_DMA  
 /\*  
 \*ZONE\_DMA标记适合DMA的内存区域，该区域的长度依赖于处理器类型，IA32一般是16M  
  \*/  
 ZONE\_DMA,  
#endif  
#ifdef CONFIG\_ZONE\_DMA32  
 /\***

**\*ZONE\_DMA标记了32位地址字可寻址，适合DMA的内存域。在32位机上本内存域是空的，即长度为0M，在AMD64或Alpha上该内存域的长度可能从0~4G。  
  \*/  
 ZONE\_DMA32,  
#endif  
 /\*  
  ZONE\_NORMAL标记了可直接映射到内核段的普通区域，可保证存在的唯一内存区域，但是无法保证都会该地址范围内对应了实际的物理内存  
  \*/  
 ZONE\_NORMAL,  
#ifdef CONFIG\_HIGHMEM   
 ZONE\_HIGHMEM, /\* 标记了超出内核段的物理内存 \*/  
#endif  
 ZONE\_MOVABLE, /\* 防止物理内存碎片的机制需要使用该内存区域 \*/  
 \_\_MAX\_NR\_ZONES /\* 充当结束标记，在内核想迭代系统中所有内存域时会用到该常量\*/  
};**

**注意:根据编译需求配置编译。例如在64位系统中不需要高端内存域，如果仅访问4G以下的外设，需要DMA内存域。**

**每个内存域都关联了一个数组，用来组织属于内存域的物理内存页，每个物理内存页都分配了一个struct\_page实例以及所需管理数据。**

**各个内存结点都保存在一个单链表中，供内核遍历。**

**内核分配内存时，总是试图运行在与当前CPU相关联的NUMA结点，但若该结点内存耗尽，则需使用每个结点提供的备用列表（struct zonelist）。**

**3.2.2 数据结构**

1. **结点数据结构**

**typedef struct pglist\_data {  
 struct zone node\_zones[MAX\_NR\_ZONES];  
 struct zonelist node\_zonelists[MAX\_ZONELISTS];  
 int nr\_zones;  
#ifdef CONFIG\_FLAT\_NODE\_MEM\_MAP /\* means !SPARSEMEM \*/  
 struct page \*node\_mem\_map;  
#ifdef CONFIG\_MEMCG  
 struct page\_cgroup \*node\_page\_cgroup;  
#endif  
#endif  
#ifndef CONFIG\_NO\_BOOTMEM  
 struct bootmem\_data \*bdata;  
#endif  
#ifdef CONFIG\_MEMORY\_HOTPLUG  
 /\*  
  \* Must be held any time you expect node\_start\_pfn, node\_present\_pages  
  \* or node\_spanned\_pages stay constant.  Holding this will also  
  \* guarantee that any pfn\_valid() stays that way.  
  \*  
  \* Nests above zone->lock and zone->size\_seqlock.  
  \*/  
 spinlock\_t node\_size\_lock;  
#endif  
 unsigned long node\_start\_pfn;  
 unsigned long node\_present\_pages; /\* total number of physical pages \*/  
 unsigned long node\_spanned\_pages; /\* total size of physical page  
          range, including holes \*/  
 int node\_id;  
 nodemask\_t reclaim\_nodes; /\* Nodes allowed to reclaim from \*/  
 wait\_queue\_head\_t kswapd\_wait;  
 wait\_queue\_head\_t pfmemalloc\_wait;  
 struct task\_struct \*kswapd; /\* Protected by lock\_memory\_hotplug() \*/  
 int kswapd\_max\_order;  
 enum zone\_type classzone\_idx;  
#ifdef CONFIG\_NUMA\_BALANCING  
 /\*  
  \* Lock serializing the per destination node AutoNUMA memory  
  \* migration rate limiting data.  
  \*/  
 spinlock\_t numabalancing\_migrate\_lock;**

**/\* Rate limiting time interval \*/  
 unsigned long numabalancing\_migrate\_next\_window;**

**/\* Number of pages migrated during the rate limiting time interval \*/  
 unsigned long numabalancing\_migrate\_nr\_pages;  
#endif  
} pg\_data\_t;**