<http://blog.chinaunix.net/uid-20543672-id-2996319.html>

# 进程栈和内核栈

在重游《LDD3》的时候，又发现了一个当年被我忽略的一句话:

“内核具有非常小的栈，它可能只和一个4096字节大小的页那样小”

### 进程堆、进程栈与线程栈

1.线程栈的空间开辟在所属进程的堆区与共享内存区之间，线程与其所属的进程共享进程的用户空间，所以线程栈之间可以互访。线程栈的起始地址和大小存放在pthread\_attr\_t 中，栈的大小并不是用来判断栈是否越界，而是用来初始化避免栈溢出的缓冲区的大小（或者说安全间隙的大小）

2.进程初始化的时候，系统会在进程的地址空间中创建一个堆，叫进程默认堆。进程中所有的线程共用这一个堆。当然，可以增加1个或几个堆，给不同的线程共同使用或单独使用。----一个进程可以多个堆

3、创建线程的时候，系统会在进程的地址空间中分配1块内存给线程栈，通常是1MB或4MB或8MB。线程栈是独立的，但是还是可以互访，因为线程共享内存空间

4.堆的分配：从操作系统角度来看，进程分配内存有两种方式，分别由两个系统调用完成：brk（）和mmap（），glibc中malloc封装了

针对这句话，我简单地学习了一下进程的“内核栈”

1. linux下的cpu有两个状态：内核态和用户态，内核态的cpu的权限高于用户态下的cpu。
2. linux下的内存分为用户态内存和内核态内存，一般4个G内存，3个G的分给用户态，1个G分给内核态。
3. linux进程有时需要调用内核资源时，如读写文件，io读写等，这时候是通过系统调用实现对内核资源的访问的，在访问内核资源前是用户栈，经过系统调用进入到内核态时，cpu的状态也由用户态变为内核态，访问的内存就是内核态下管理的内存了-内核栈，对内核里的资源访问完返回，内存又回到了用户栈，cpu也回到用户态。
4. 当进程因为中断或者系统调用而陷入内核态之行时，进程所使用的堆栈也要从用户栈转到内核栈。

### 什么是进程的“内核栈”？

在每一个进程的生命周期中，必然会通过到系统调用陷入内核。在执行系统调用陷入内核之后，这些内核代码所使用的栈并不是原先用户空间中的栈，而是一个内核空间的栈，这个称作进程的“内核栈”。

比如，有一个简单的字符驱动实现了open方法。在这个驱动挂载后，应用程序对那个驱动所对应的设备节点执行open操作，这个应用程序的open其实就通过glib库调用了Linux的open系统调用，执行系统调用陷入内核后，处理器转换为了特权模式（**具体的转换机制因构架而异，对于ARM来说普通模式和用户模式的的栈针（SP）是不同的寄存器**），此时使用的栈指针就是内核栈指针，他指向内核为每个进程分配的内核栈空间。

### 内核栈的作用

我个人的理解是：在陷入内核后，系统调用中也是存在函数调用和自动变量，这些都需要栈支持。用户空间的栈显然不安全，需要内核栈的支持。此外，内核栈同时用于保存一些系统调用前的应用层信息（如用户空间栈指针、系统调用参数）。

### 内核栈与进程结构体的关联

每个进程在创建的时候都会得到一个内核栈空间，内核栈和进程的对应关系是通过2个结构体中的指针成员来完成的：

#### （1）struct task\_struct

在学习Linux进程管理肯定要学的结构体，在内核中代表了一个进程，其中记录的进程的所有状态信息，定义在Sched.h (include\linux)。

其中有一个成员：void \*stack;就是指向下面的内核栈结构体的“栈底”。

在系统运行的时候，宏current获得的就是当前进程的struct task\_struct结构体。

#### （2）内核栈结构体union thread\_union

union thread\_union {

struct thread\_info thread\_info;

unsigned long stack[THREAD\_SIZE/sizeof(long)];

};

其中struct thread\_info是记录部分进程信息的结构体，其中包括了进程上下文信息:

/\*

\* low level task data that entry.S needs immediate access to.

\* \_\_switch\_to() assumes cpu\_context follows immediately after cpu\_domain.

\*/

struct thread\_info {

unsigned long flags; /\* low level flags \*/

int preempt\_count; /\* 0 => preemptable, <0 => bug \*/

mm\_segment\_t addr\_limit; /\* address limit \*/

**struct task\_struct \*task; /\* main task structure \*/**

struct exec\_domain \*exec\_domain; /\* execution domain \*/

\_\_u32 cpu; /\* cpu \*/

\_\_u32 cpu\_domain; /\* cpu domain \*/

struct cpu\_context\_save cpu\_context; /\* cpu context \*/

\_\_u32 syscall; /\* syscall number \*/

\_\_u8 used\_cp[16]; /\* thread used copro \*/

unsigned long tp\_value;

struct crunch\_state crunchstate;

union fp\_state fpstate \_\_attribute\_\_((aligned(8)));

union vfp\_state vfpstate;

#ifdef CONFIG\_ARM\_THUMBEE

unsigned long thumbee\_state; /\* ThumbEE Handler Base register \*/

#endif

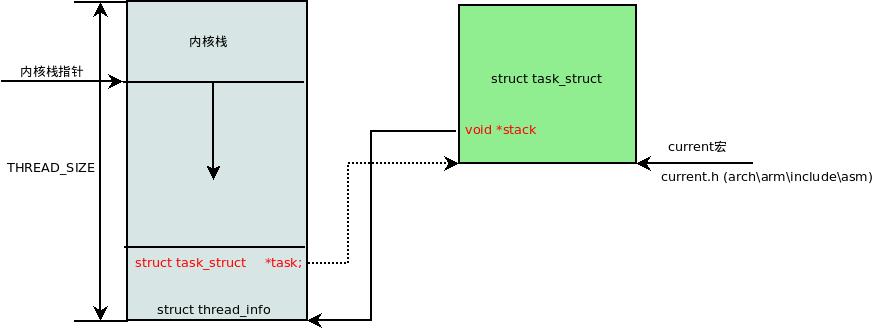
struct restart\_block restart\_block;

};

关键是其中的task成员，指向的是所创建的进程的struct task\_struct结构体

而其中的stack成员就是内核栈。从这里可以看出内核栈空间和 thread\_info是共用一块空间的。如果内核栈溢出， thread\_info就会被摧毁，系统崩溃了～～～

内核栈---struct thread\_info----struct task\_struct三者的关系入下图：



### 内核栈的产生

在进程被创建的时候，fork族的系统调用中会分别为内核栈和struct task\_struct分配空间，调用过程是：

**fork族的系统调用--->do\_fork--->copy\_process--->dup\_task\_struct**

在dup\_task\_struct函数中：

static struct task\_struct \*dup\_task\_struct(struct task\_struct \*orig)

{

struct task\_struct \*tsk;

struct thread\_info \*ti;

unsigned long \*stackend;

**int err;**

prepare\_to\_copy(orig);

**tsk = alloc\_task\_struct();**

if (!tsk)

return NULL;

**ti = alloc\_thread\_info(tsk);**

if (!ti) {

free\_task\_struct(tsk);

return NULL;

}

**err = arch\_dup\_task\_struct(tsk, orig);**

if (err)

goto out;

**tsk->stack = ti;**

err = prop\_local\_init\_single(&tsk->dirties);

if (err)

goto out;

**setup\_thread\_stack(tsk, orig);**

......

其**中alloc\_task\_struct使用内核的**slab分配器去为所要创建的进程分配struct task\_struct的空间

**而alloc\_thread\_info使用内核的**伙伴系统去为所要创建的进程分配内核栈（union thread\_union ）空间

注意：

**后面的tsk->stack = ti;语句，这就是关联了struct task\_struct和内核栈**

**而在setup\_thread\_stack(tsk, orig);中，关联了内核栈和struct task\_struct：**

static inline void setup\_thread\_stack(struct task\_struct \*p, struct task\_struct \*org)

{

\*task\_thread\_info(p) = \*task\_thread\_info(org);

task\_thread\_info(p)->task = p;

}

### 内核栈的大小

由于是每一个进程都分配一个内核栈空间，所以不可能分配很大。这个大小是构架相关的，一般以页为单位。其实也就是上面我们看到的THREAD\_SIZE，这个值一般为4K或者8K。对于ARM构架，这个定义在Thread\_info.h (arch\arm\include\asm)，

#define THREAD\_SIZE\_ORDER 1

#define THREAD\_SIZE 8192

#define THREAD\_START\_SP (THREAD\_SIZE - 8)

**所以ARM的内核栈是8KB**

**在（内核）驱动编程时需要注意的问题：**

**由于栈空间的限制，在编写的驱动（特别是被系统调用使用的底层函数）中要注意避免对栈空间消耗较大的代码，比如递归算法、局部自动变量定义的大小等等**

更多关于内核栈的资料请参考：

[Linux内核栈溢出（stack overflow）问题](http://hi.baidu.com/god_sperm/blog/item/511059f73a9250f77709d769.html)

[Linux内核2.6和2.4中内核堆栈的比较](http://feizf.blogbus.com/logs/16835565.html)

[4.4.1进程内核栈](http://oss.org.cn/kernel-book/ch04/4.4.1.htm)

[内核栈的大小](http://hi.baidu.com/whs08/blog/item/84437202d89a35074bfb5196.html)

[专题研究一  进程的深入理解与分析](http://blog.chinaunix.net/space.php?uid=20543672&do=blog&id=2998608)(必看)

# Linux内核中的中断栈与内核栈的补充说明

中断栈与内核栈的话题更多地属于内核的范畴，所以在《深入Linux设备驱动程序内核机制》第5章“中断处理”当中，基本上没怎么涉及到上述内容，只是在5.4节有些许的文字讨论中断栈在中断嵌套情形下可能的溢出问题。

本贴在这个基础上对内核栈与中断栈的话题做些补充，讨论基于x86 32位系统，因为64位系统下Linux内核关于栈的支持原理上是相同的，不过也有些特性属于64位特有的，比如IST(Interrupt Stack Table)，如果可能将来会在processor版块发个帖子专门讨论。

现在的Linux内核中采用的是内核栈与中断栈分离的设计，下面我们从源码层面来看一看这种分离是如何完成的。

1. **x86下内核栈与中断栈是否共享的问题**

内核栈与中断栈分离的核心代码发生在do\_IRQ() --> handle\_irq() --> execute\_on\_irq\_stack()最后一个函数字面上的意思大约是在中断栈中执行中断处理例程，也就是说中断的处理函数会在独立于被中断进程的上下文中执行。execute\_on\_irq\_stack的函数实现为：

static inline int

execute\_on\_irq\_stack(int overflow, struct irq\_desc \*desc, int irq)

{

union irq\_ctx \*curctx, \*irqctx;

u32 \*isp, arg1, arg2;

curctx = (union irq\_ctx \*) current\_thread\_info();

irqctx = \_\_this\_cpu\_read(hardirq\_ctx);

/\*

\* this is where we switch to the IRQ stack. However, if we are

\* already using the IRQ stack (because we interrupted a hardirq

\* handler) we can't do that and just have to keep using the

\* current stack (which is the irq stack already after all)

\*/

if (unlikely(curctx == irqctx))

return 0;

/\* build the stack frame on the IRQ stack \*/

isp = (u32 \*) ((char \*)irqctx + sizeof(\*irqctx));

irqctx->tinfo.task = curctx->tinfo.task;

irqctx->tinfo.previous\_esp = current\_stack\_pointer;

/\*

\* Copy the softirq bits in preempt\_count so that the

\* softirq checks work in the hardirq context.

\*/

irqctx->tinfo.preempt\_count =

(irqctx->tinfo.preempt\_count & ~SOFTIRQ\_MASK) |

(curctx->tinfo.preempt\_count & SOFTIRQ\_MASK);

if (unlikely(overflow))

call\_on\_stack(print\_stack\_overflow, isp);

asm volatile("xchgl %%ebx,%%esp \n"

"call \*%%edi \n"

"movl %%ebx,%%esp \n"

: "=a" (arg1), "=d" (arg2), "=b" (isp)

: "0" (irq), "1" (desc), "2" (isp),

"D" (desc->handle\_irq)

: "memory", "cc", "ecx");

return 1;

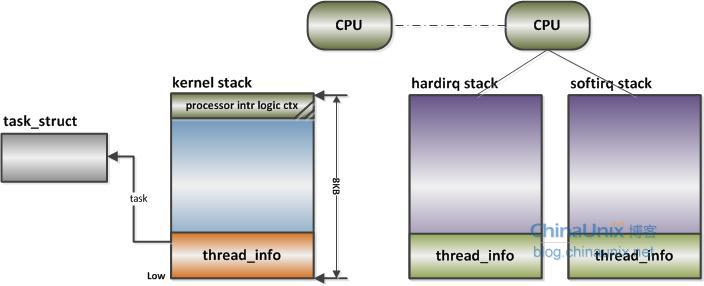
}

代码中的curctx=(union irq\_ctx \*) current\_thread\_info()用来获得当前被中断进程的上下文，irqctx = \_\_this\_cpu\_read(hardirq\_ctx)用来获得hardirq的上下文，其实就是获得独立的中断栈起始地址。中断栈的大小与layout与内核栈是完全一样的。接下来isp指向中断栈栈顶，最后的堆栈切换发生在那段汇编代码中：当前进程的内核栈ESP指针保存在EBX中，而中断栈的isp则赋值给了ESP，这样接下来的代码就将使用中断栈了。call语句负责调用desc->handle\_irq()函数，这里会进行中断处理，设备驱动程序注册的中断处理函数会被调用到。当中断处理例程结束返回时，ESP将重新指向被中断进程的内核栈。(此处我们应该注意到内核栈中还保留着中断发生时处理器硬件逻辑所压入的CS, EIP等寄存器，所以在内核栈中做中断返回是完全正确的）。

2. 中断栈的分配

独立的中断栈所在内存空间的分配发生在arch/x86/kernel/irq\_32.c的irq\_ctx\_init函数中(如果是多处理器系统，那么每个处理器都会有一个独立的中断栈)，函数使用\_\_alloc\_pages在低端内存区分配2个物理页面(2的THREAD\_ORDER次方)，也就是8KB大小的空间。有趣的是，这个函数还会为softirq分配一个同样大小的独立堆栈，如此说来，softirq将不会在hardirq的中断栈上执行，而是在自己的上下文中执行。

总结一下，系统中每个进程都会拥有属于自己的内核栈，而系统中每个CPU都将为中断处理准备了两个独立的中断栈，分别是hardirq栈和softirq栈。草图如下：



关于在中断处理函数中涉及到的阻塞问题，我个人的观点是：现实中绝对不要这么干，其中的原因就不多说了。从内核理论实现的角度，调度其他进程是可行的。

<http://blog.chinaunix.net/uid-20543672-id-3164600.html>

# Linux独立中断栈学习笔记及验证实验（ARM、x86）

对于IA-32平台，高版本的内核（>2.6.36）内核都实现了独立的中断栈

对于ARM平台，内核暂未实现独立的中断栈

PS：其实对于独立的中断栈，不仅在IA-32上有，PPC等其他平台也有实现，有兴趣的朋友可以参考内核源码。对于上面的实验，也可以使用软件中断（如tasklet）再做一次，有兴趣的朋友可以自行修改源码。

http://19880512.blog.51cto.com/936364/274610

# Linux 系统中堆栈的使用方法

Linux 0.12系统中共使用了4种堆栈。

第1种是系统引导初始化时临时使用的堆栈；

第2种是进入保护模式之后提供内核程序初始化使用的堆栈，位于内核代码地址空间固定位置处。该堆栈也是后来**任务0**使用的用户态堆栈；

第3种是每个任务通过系统调用，执行内核程序时使用的堆栈，我们称之为任务的内核态堆栈。每个任务都有自己独立的内核态堆栈；

第4种是任务在用户态执行的堆栈，位于任务（进程）逻辑地址空间近末端处。

使用多个栈或在不同情况下使用不同栈的主要原因有两个。首先是由于从实模式进入保护模式，使得CPU对内存寻址访问方式发生了变化，因此需要重新调整设置栈区域。另外，为了解决不同CPU特权级共享使用堆栈带来的保护问题，执行0级的内核代码和执行3级的用户代码需要使用不同的栈。当一个任务进入内核态运行时，就会使用其TSS段中给出的特权级0的堆栈指针tss.ss0、tss.esp0，即内核栈。原用户栈指针会被保存在内核栈中。而当从内核态返回用户态时，就会恢复使用用户态的堆栈。下面分别对它们进行说明。

## 初始化阶段

（1）开机初始化时(bootsect.S，setup.s)

当bootsect代码被ROM BIOS引导加载到物理内存0x7c00处时，并没有设置堆栈段，当然程序也没有使用堆栈。直到bootsect被移动到0x9000:0处时，才把堆栈段寄存器SS设置为0x9000，堆栈指针esp寄存器设置为0xff00，**即堆栈顶端在0x9000:0xff00**处，参见boot/bootsect.s第61、62行。setup.s程序中也沿用了bootsect中设置的堆栈段。这就是系统初始化时临时使用的堆栈。

（2）进入保护模式时(head.s)

从head.s程序起，系统开始正式在保护模式下运行。此时堆栈段被设置为内核数据段（0x10），堆栈指针esp设置成指向user\_stack数组的顶端（参见head.s，第31行），保留了1页内存（4KB）作为堆栈使用。user\_stack数组定义在sched.c的67～72行，共含有1024个长字。它在物理内存中的位置示意图可参见图5-23。此时该堆栈是内核程序自己使用的堆栈。其中给出的地址是大约值，它们与编译时的实际设置参数有关。这些地址位置是从编译内核时生成的system.map文件中查到的。

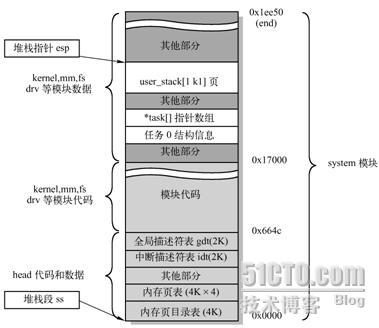


图5-23 刚进入保护模式时内核使用的堆栈示意图

（3）初始化时(main.c)

在init/main.c程序中，在执行**move\_to\_user\_mode()**代码把控制权移交给任务0之前，系统一直使用上述堆栈。而在执行过move\_to\_user\_mode()之后，main.c的代码被“切换”成任务0中执行。通过执行fork()系统调用，main.c中的init()将在任务1中执行，并使用任务1的堆栈。而main()本身则在被“切换”成为任务0后，仍然继续使用上述内核程序自己的堆栈作为任务0的用户态堆栈。关于任务0所使用堆栈的详细描述见后面说明。

## 任务的堆栈

每个任务都有两个堆栈，分别用于用户态和内核态程序的执行，并且分别称为用户态堆栈和内核态堆栈。除了处于不同CPU特权级中，这两个堆栈之间的主要区别在于任务的内核态堆栈很小，所保存的数据量最多不能超过4096 – 任务数据结构块个字节，大约为3KB。而任务的用户态堆栈却可以在用户的64MB空间内延伸。

（1）在用户态运行时

每个任务（除了任务0和任务1）有自己的64MB地址空间。当一个任务（进程）刚被创建时，它的用户态堆栈指针被设置在其地址空间的靠近末端（64MB顶端）部分。实际上末端部分还要包括执行程序的参数和环境变量，然后才是用户堆栈空间，如图5-24所示。应用程序在用户态下运行时就一直使用这个堆栈。堆栈实际使用的物理内存则由CPU分页机制确定。由于Linux实现了写时复制功能（Copy on Write），因此在进程被创建后，若该进程及其父进程都没有使用堆栈，则两者共享同一堆栈对应的物理内存页面。只有当其中一个进程执行堆栈写操作（如push操作）时内核内存管理程序才会为写操作进程分配新的内存页面。而进程0和进程1的用户堆栈比较特殊，见后面说明。



图5-24 逻辑空间中的用户态堆栈

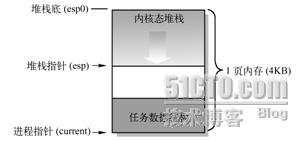
（2）在内核态运行时

每个任务都有自己的内核态堆栈，用于任务在内核代码中执行期间。其所在线性地址中的位置由该任务TSS段中ss0和esp0两个字段指定。ss0是任务内核态堆栈的段选择符，esp0是堆栈栈底指针。因此每当任务从用户代码转移进入内核代码中执行时，任务的内核态栈总是空的。任务内核态堆栈被设置在位于其任务数据结构所在页面的末端，即与任务的任务数据结构（task\_struct）放在同一页面内。这是在建立新任务时，fork()程序在任务tss段的内核级堆栈字段（tss.esp0和tss.ss0）中设置的，参见kernel/fork.c，92行：

p->tss.esp0 = PAGE\_SIZE + (long)p;

p->tss.ss0 = 0x10;

其中，p是新任务的任务数据结构指针，tss是任务状态段结构。内核为新任务申请内存用作保存其task\_struct结构数据，而tss结构（段）是task\_struct中的一个字段。该任务的内核堆栈段值tss.ss0也被设置成为0x10（即内核数据段选择符），而tss.esp0则指向保存task\_struct结构页面的末端。如图5-25所示。实际上tss.esp0被设置成指向该页面（外）上一字节处（图中堆栈底处）。这是因为Intel CPU执行堆栈操作时是先递减堆栈指针esp值，然后在esp指针处保存入栈内容。



（3）任务0和任务1的堆栈

任务0（空闲进程idle）和任务1（初始化进程init）的堆栈比较特殊，需要特别予以说明。任务0和任务1的代码段和数据段相同，限长也都是640KB，但它们被映射到不同的线性地址范围中。任务0的段基地址从线性地址0开始，而任务1的段基地址从64MB开始。但是它们全都映射到物理地址0～640KB范围中。这个地址范围也就是内核代码和基本数据所存放的地方。在执行了move\_to\_user\_mode()之后，任务0和任务1的内核态堆栈分别位于各自任务数据结构所在页面的末端，而任务0的用户态堆栈就是前面进入保护模式后所使用的堆栈，即sched.c的user\_stack[]数组的位置。由于任务1在创建时复制了任务0的用户堆栈，因此刚开始时任务0和任务1共享使用同一个用户堆栈空间。但是当任务1开始运行时，由于任务1映射到user\_stack[]处的页表项被设置成只读，使得任务1在执行堆栈操作时将会引起写页面异常，从而内核会使用写时复制机制（关于写时复制技术的说明请参见第13章）为任务1另行分配主内存区页面作为堆栈空间使用。只有到此时，任务1才开始使用自己独立的用户堆栈内存页面。因此任务0的堆栈需要在任务1实际开始使用之前保持“干净”，即任务0此时不能使用堆栈，以确保复制的堆栈页面中不含有任务0的数据。

# 任务0

1. 进程0是所有其他进程的祖先, 也称作idle进程或swapper进程.

2. 进程0是在系统初始化时由kernel自身从无到有创建.

3. 进程0的数据成员大部分是静态定义的，即由预先定义好的INIT\_TASK, INIT\_MM等宏初始化.

进程0的描述符init\_task定义在arch/arm/kernel/init\_task.c,由INIT\_TASK宏初始化。 init\_mm等结构体定义在include/linux/init\_task.h内，为init\_task成员的初始值,分别由对应的初始化宏如INIT\_MM等初始化

<http://blog.chinaunix.net/uid-25871104-id-3073916.html>

asmlinkage void \_\_init start\_kernel(void)

rest\_init();

kernel\_thread(kernel\_init, NULL, CLONE\_FS | CLONE\_SIGHAND);

**cpu\_idle();**

## cpu\_idle

/arch/x86/kernel/process\_32.c

**void cpu\_idle(void)**

**{**

**/\* endless idle loop with no priority at all \*/**

**while (1) {**

**stop\_critical\_timings();**

**pm\_idle();**

**start\_critical\_timings();**

**}**

}

### pm\_idle

II，cpu\_idle driver而cpu\_idle()函数的核心是pm\_idle()函数。pm\_idle()函数是一个全局变量，它首先会在identify\_cpu()函数中，即系统识别CPU时被赋值，默认是default\_dile()函数。接下来cpu\_idle这支driver会根据CPU的类型和内核中存在的idle driver来重新设定pm\_idle全局变量。

**确切的说，cpu\_idle是一个驱动框架，在此框架下，不同的idle驱动可以根据硬件不同和需求不同来编写不同进入和退出Cx的行为。**

1. 核心数据结构首先是cpuidle\_state,

struct cpuidle\_state {

char name[CPUIDLE\_NAME\_LEN];

char desc[CPUIDLE\_DESC\_LEN];

void \*driver\_data;

unsigned int flags;

unsigned int exit\_latency; /\* in US \*/

unsigned int power\_usage; /\* in mW \*/

unsigned int target\_residency; /\* in US \*/

unsigned long long usage;

unsigned long long time; /\* in US \*/

int (\*enter) (struct cpuidle\_device \*dev,

struct cpuidle\_state \*state);

};

该结构描述CPU的一个Cx state，最主要的就是enter，即该state的进入函数，另外也有该state的退出延迟，消耗电能等。

然后是cpuidle\_device,

struct cpuidle\_device {

unsigned int registered:1;

unsigned int enabled:1;

unsigned int power\_specified:1;

unsigned int cpu;

int last\_residency;

int state\_count;

struct cpuidle\_state states[CPUIDLE\_STATE\_MAX];

struct cpuidle\_state\_kobj \*kobjs[CPUIDLE\_STATE\_MAX];

struct cpuidle\_state \*last\_state;

struct list\_head device\_list;

struct kobject kobj;

struct completion kobj\_unregister;

void \*governor\_data;

struct cpuidle\_state \*safe\_state;

int (\*prepare) (struct cpuidle\_device \*dev);

};

该结构描述了cpu所具有的Cx states特性，包括其支持的states列表:statas[]，上一个state状态:\*last\_state，最安全的state:\*safe\_state。

另外，还有一个注册driver用的cpuidle\_driver，

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* CPUIDLE DRIVER INTERFACE \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

struct cpuidle\_driver {

char name[CPUIDLE\_NAME\_LEN];

struct module \*owner;

};

2，暴露给idle driver的接口2.1 cpuidle\_register\_device(struct cpuidle\_device \*dev)idle driver使用该接口将struct cpuidle\_device注册到cpu\_idle框架中。如果注册成功，在代码的最后还会调用cpuidle\_install\_idle\_handler()函数，将上面提到的pm\_idle全局变量赋为cpuidle\_idle\_call()。值得注意的是，cpuidle中存在一个全局变量pm\_idle\_old，保存之前pm\_idle值，当idle driver被卸载后，pm\_idle会被回复成原来的值。

2.2 cpuidle\_register\_driver(struct cpuidle\_driver \*drv)idle driver使用该接口将struct cpuidle\_driver注册到cpu\_idle框架中。查看cpuidle\_register\_driver的代码可知，cpuidle不允许多支idle driver同时运行。

3，核心流程上面提到pm\_idle全局量在cpuidle\_register\_device()函数中被赋为cpuidle\_idle\_call()，那么每次idle被调度(准确的说idle进程并不参与调度，而是当可运行进程列表为空时被运行)，都会进入cpuidle\_idle\_call()函数。

参考网址：http://siyobik.info/main/reference/instruction/MONITOR

http://siyobik.info/main/reference/instruction/MWAIT

按照oracal这这篇文章来看，这种进入Cx的方式有较低的唤醒延迟。

http://docs.oracle.com/cd/E19082-01/820-2604/ggdqv/index.html

值得注意的是，如果内核使用intel\_idle，即使你通过BIOS或者内核参数关闭ACPI，或者禁用了某个Cx state，

intel\_idle一样会工作。如果要禁用intel\_idle，可使用内核参数"intel\_idle.max\_cstate=0"

网上有人碰到这样的问题：

机器响应很迟钝，打开ACPI时，发现是acpi\_idle\_enter\_bm()吃掉了大多数时间，关闭ACPI，发现是mwait\_idle()吃掉了大多数时间。

<http://www.spinics.net/lists/linux-acpi/msg24058.html>

# 任务1

Process 1

进程0最终会通过调用kernel\_thread创建一个内核线程去执行init函数，这个新创建的内核线程即Process 1(这时还是共享着内核线程0的资源属性如地址空间等)。init函数继续完成剩余的内核初始化,并在函数的最后调用execve系统调用装入用户空间的可执行程序/sbin/init,这时进程1就拥有了自己的属性资源，成为一个普通进程(init进程)。至此，内核初始化和启动过程结束。下面就进入了用户空间的初始化，最后运行shell登陆界面。(注：Init进程一直存活，因为它创建和监控在操作系统外层执行的所有进程的活动。)

# Kthreadd

kernel\_thread(kthreadd, NULL, CLONE\_FS | CLONE\_FILES);

int kthreadd(void \*unused)

kthreadd的核心是一for和while循环体。在for循环中，如果发现kthread\_create\_list是一空链表，则调用 schedule调度函数，因为此前已经将该进程的状态设置为TASK\_INTERRUPTIBLE，所以schedule的调用将会使当前进程进入睡 眠。如果kthread\_create\_list不为空，则进入while循环，在该循环体中会遍历该kthread\_create\_list列表，对于 该列表上的每一个entry，都会得到对应的类型为struct kthread\_create\_info的节点的指针create.

然后函数在kthread\_create\_list中删除create对应的列表entry，接下来以create指针为参数调用create\_kthread(create).

在create\_kthread()函数中，会调用kernel\_thread来生成一个新的进程，该进程的内核函数为kthread，调用参数为create:

kernel\_thread(kthread, create, CLONE\_FS | CLONE\_FILES | SIGCHLD);

## kthread\_create和kernel\_thread

kthread\_create是通过work\_queue来实现的，kernel\_thread是通过do\_fork来实现的。

kernel thread可以用kernel\_thread创建，但是在执行函数里面必须用daemonize释放资源并挂到init下，还需要用 completion等待这一过程的完成。

kthread\_create是比较正牌的创建函数，这个不必要调用daemonize，用这个创建的kernel thread都挂在了kthread线程下。

可以在非内核线程中调用kernel\_thread, 但这样创建的线程必须在自己调用daemonize(...)来释放资源，成为真正的内核线程。

### kthread\_create

1 使用kthread\_create创建线程：

struct task\_struct \*kthread\_create(int (\*threadfn)(void \*data),

void \*data,

const char \*namefmt, ...);

这个函数可以像printk一样传入某种格式的线程名

线程创建后，不会马上运行，而是需要将kthread\_create() 返回的task\_struct指针传给wake\_up\_process()，然后通过此函数运行线程。

2. 当然，还有一个创建并启动线程的函数：kthread\_run

struct task\_struct \*kthread\_run(int (\*threadfn)(void \*data),

void \*data,

const char \*namefmt, ...);

3. 线程一旦启动起来后，会一直运行，除非该线程主动调用do\_exit函数，或者其他的进程调用kthread\_stop函数，结束线程的运行。

int kthread\_stop(struct task\_struct \*thread);

kthread\_stop() 通过发送信号给线程。

如果线程函数正在处理一个非常重要的任务，它不会被中断的。当然如果线程函数永远不返回并且不检查信号，它将永远都不会停止。

### API

struct task\_struct kthread\_run(int (\*threadfn)(void \*data), void \*data, const char namefmt[],...);

int kthread\_stop(struct task\_struct \*k);

int kthread\_should\_stop(void);