Understanding the Linux Kernel

对照2.6.11内核，可以随意转载。作者：金庆辉

Chapter 2. Memory Addressing

2.1. Memory Addresses

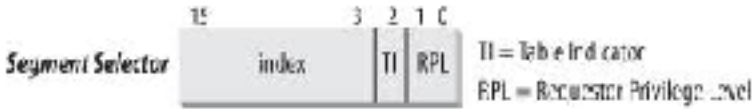
Figure 2-1. Logical address translation

2.2. Segmentation in Hardware

2.2.1. Segment Selectors and Segmentation Registers

一个逻辑地址由两部分组成：

1. segment identifier：16bit，叫**Segment Selector**



2. Offset:32bit。

6个段寄存器存放Segment Selectors，cs，ss，ds，es，fs，和gs。
其中：
cs:代码段。 ss:堆栈段。 ds:数据段。
剩下3个段寄存器是通用的，可以存放任意数据段。
cs有2bit指定了Current Privilege Level (CPL)。

2.2.2. Segment Descriptors

每个段都被一个8字节的Segment Descriptor描述。Segment Descriptor存放在Global Descriptor Table (GDT) or in the Local Descriptor Table(LDT)中。通常只适用GDT，除非进程想创建额外的segment。
gdtr和ldtr控制寄存器。

Segment Descriptor: DPL -限制了访问该段的最小CPU等级（CS中的CPL）权限。比如，DPL为0，则只有当CPL为0时才能访问。

四种最常用的Segment Descriptors(存放在GDT中):

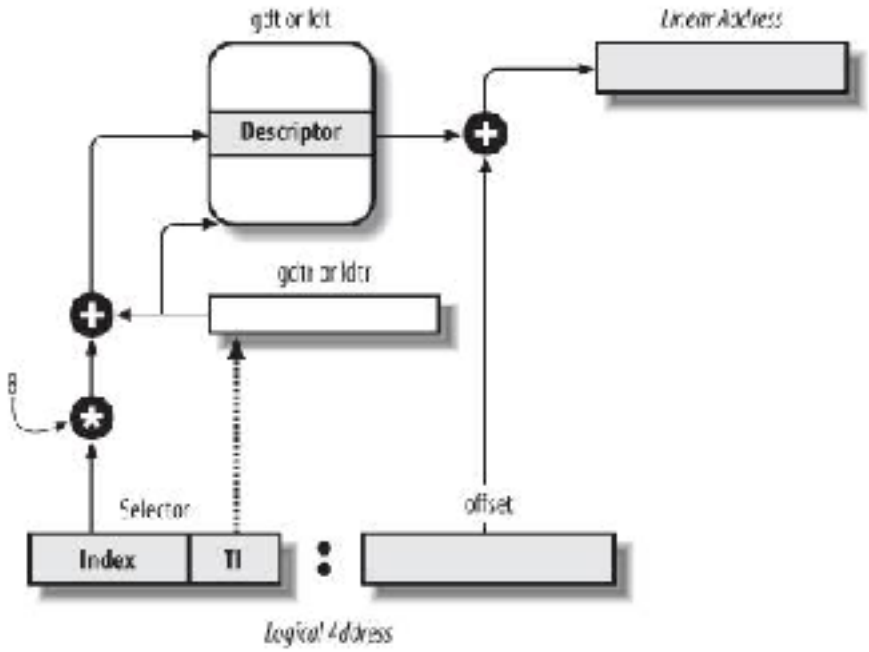
1. Code Segment Descriptor
2. Date Segment Descriptor: Linux中代码段堆，栈段都描述符一样。
3. Task State Segment Descriptor (TSSD): 指向Task State Segment (TSS)，TSS用来存放进程寄存器内容，TSS只在GDT中。
4. Local Descriptor Table Descriptor (LDTD): LDT的描述符。

2.2.3. Fast Access to Segment Descriptors

每个段寄存器都有一个对应的不可编程的"影子"寄存器(8字节)，当段寄存器加载Segment Selector时，Segment Selector对应在GDT中的Segment Descriptors就会放入这个"影子"寄存器。

GDT第一项是全0，这样当访问逻辑地址0时，就会产生异常。GDT可用项最大为8,191(2的13次方-1).

2.2.4. Segmentation Unit



2.3. Segmentation in Linux

因为段寄存器的存在，linux逻辑地址不在包含Segment Selector，而只有offset。

2.3.1. The Linux GDT

多处理器体系中，有多个GDT，每个CPU一个GDT。存放在init_tss数组中。

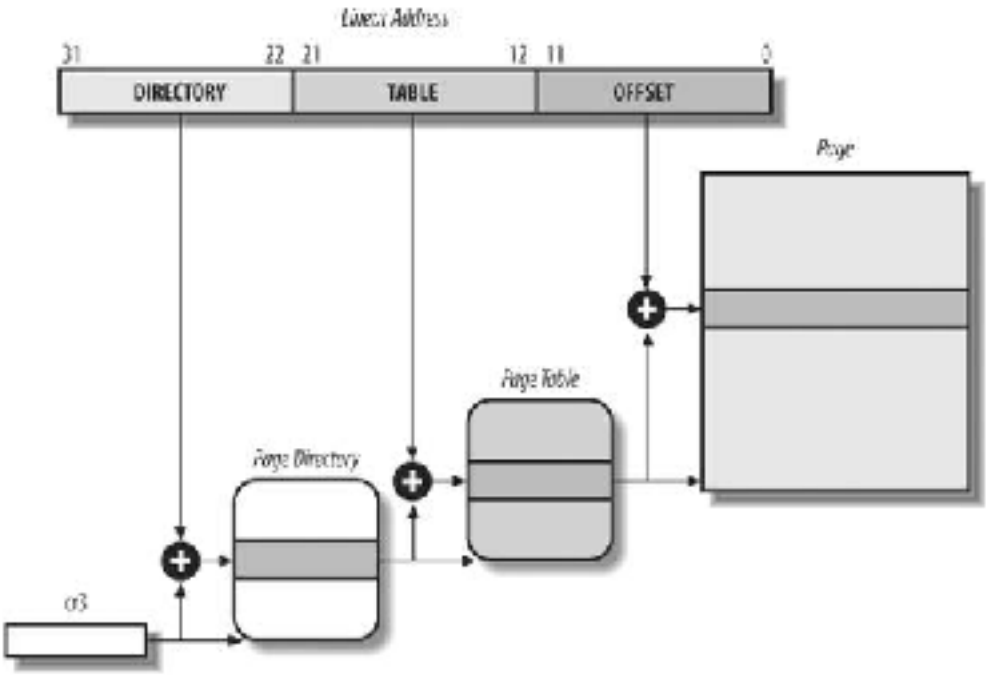
2.3.2. The Linux LDTs

Linux很少使用LDT,因此在所有GDT中指向一个缺省LDT.

2.4. Paging in Hardware

2.4.1. Regular Paging

Figure 2-7. Paging by 80 x 86 processors



Page Directory地址存放在cr3寄存器中。
Page Directory 和Page Tables结构相似:

1. Present flag: 如果访问一个Present标志位为0页面，产生页面异常，引起异常的线性地址存放在cr2寄存器中。
2. Accessed flag: 每次paging unit访问页面时置位，该位用于OS选择swap页面。paging unit不能将它复位，只有OS能。
3. Dirty flag: 对页面写操作后置位，与Accessed flag 一起用于OS选择swap的页面。paging unit不能将它复位，只有OS能。
4. Read/Write flag:确定page或Page Table的读写权限。
5. User/Supervisor flag: 确定存取page或Page Table特权等级。
6. PCD and PWT flags: 是否cache。
7. Global flag:略。

2.4.3. Hardware Protection Scheme

Read/Write flag: 0—只读，1—可读写。
User/Supervisor flag: 0--CPL为内核态才可访问。

2.4.7. Hardware Cache

物理地址对应的内存数据的缓存。

2.4.8. Translation Lookaside Buffers (TLB)

线性地址与物理地址对应值之间的缓存。

2.5. Paging in Linux

从2.6.11开始，Linux使用4层映射。
Page Global Directory
Page Upper Directory
Page Middle Directory
Page Table

2.5.5. Kernel Page Tables

内核有专用的页表，叫master kernel Page Global Directory。系统初始化后，这个页表不被内核线程直接使用。

2.5.5.3. Final kernel Page Table when RAM size is between 896 MB and 4096 MB

内核线性地址空间只能映射到896M（低端内存）。

2.5.5.4. Final kernel Page Table when RAM size is more than 4096 MB

当系统物理内存超过4G时,必须使用CPU的扩展分页(PAE)模式所提供的64位页目录项才能存取到4G以上的物理内.在PAE模式下,线性地址到物理地址的转换使用3级页表,第1级页目录由线性地址的最高2位索引， 每一目录项对应1G的寻址空间,第2级页目录项以9位索引,每一目录项对应2M的寻址空间， 第3级页目录项以9位索引,每一目录项对应4K的页帧.除了页目录项所描述的物理地址扩展为36位外,64位和32位页目录项结构没有什么区别.在PAE模式下,包含PSE位的中级页目录项所对应的页面从4M减少为2M.

2.5.6. Fix-Mapped Linear Addresses

由fixed_addresses枚举出的一些线性地址，属内核空间的顶端，专门用来映射到指定的任意物理地址。

2.5.7.2. Handling the TLB

TLB同步是由OS来决定的。原则上，当内核将新Page Global Directory地址写入cr3寄存器时，TLB刷新。

当多个CPU使用同一个Page Table时，当TLB必须在这些CPU上刷新时，对那些运行在内核线程的CPU可以使用lazy TLB模式，将TLB刷新延迟。原理是：运行在内核线程的CPU，不关心用户地址空间映射的变动，因此对，用户空间的改动，可以延时做出反应。当内核线程切换到一个用户进程（使用不同Page Table），因为使用了不同Page Table， TLB必须完全更新，如果换到一个用户进程（使用相同Page Table），失效的TLB特定项就得刷新了。

Chapter 3. Processes

3.2.1. Process State

TASK_RUNNING
TASK_INTERRUPTIBLE：硬中断，释放某资源，信号可以唤醒。

TASK_UNINTERRUPTIBLE：硬中断，释放某资源可以唤醒。
TASK_STOPPED
TASK_TRACED
EXIT_ZOMBIE:中止的进程，但还没有被父进程回收。
EXIT_DEAD：完全死去的进程。

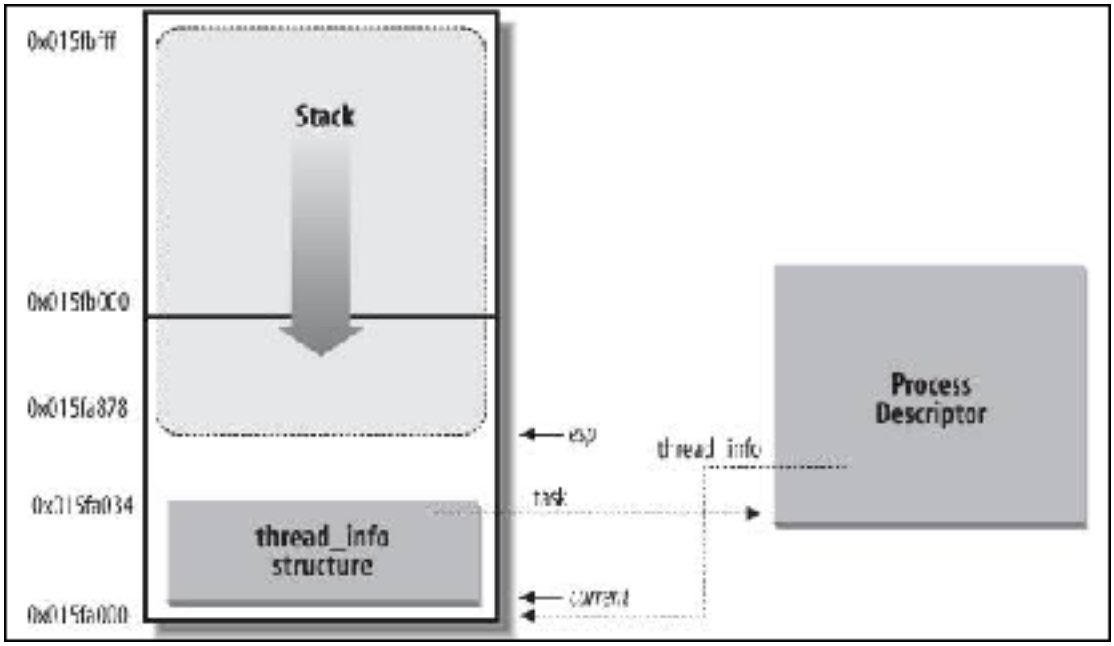
3.2.2. Identifying a Process

每个进程有各自的PID，存放在task_struct->pid中。
多线程应用中，每个线程有各自的PID,但有相同Group PID(等于第一个线程的PID)。
存放在task_struct->tpid，用户态的getpid( )函数返回的是Group PID。

3.2.2.1. Process descriptors handling

```
union thread_union {
  struct thread_info thread_info;
  unsigned long stack[2048]; /* 1024 for 4KB stacks */
};
```

见下图：



3.2.2.2. Identifying the current process

esp寄存器存放着内核栈栈顶指针，因此很容易就就可以得到thread_union地址。
在SMP下current应该是个数组元素，对应每一个CPU。

3.2.2.4. The process list

0号进程(swapper)--init_task是所有进程的头。通过task_struct->**tasks**链入。

3.2.2.5. The lists of TASK_RUNNING processes

如果进程优先级为k(0-139)，则它通过task_struct->**run_list**链入优先级为k的运行队列。

prio_array_t结构中:

struct list_head [140]　　queue　The 140 heads of the priority lists

3.2.3. Relationships Among Processes

1号进程(init)是所有其它进程的祖先。

real_parent:进程P的创造者或P的创造者退出后，被指定为1号进程。
parent:进程退出后，退出信号的的接收者。一般等于real_parent，除非是被ptrace( )。
children
sibling
group_leader: 进程组leader进程
signal->pgrp: PID of the group leader of P
tgid: PID of the thread group leader of P

signal->session: PID of the login session leader of P

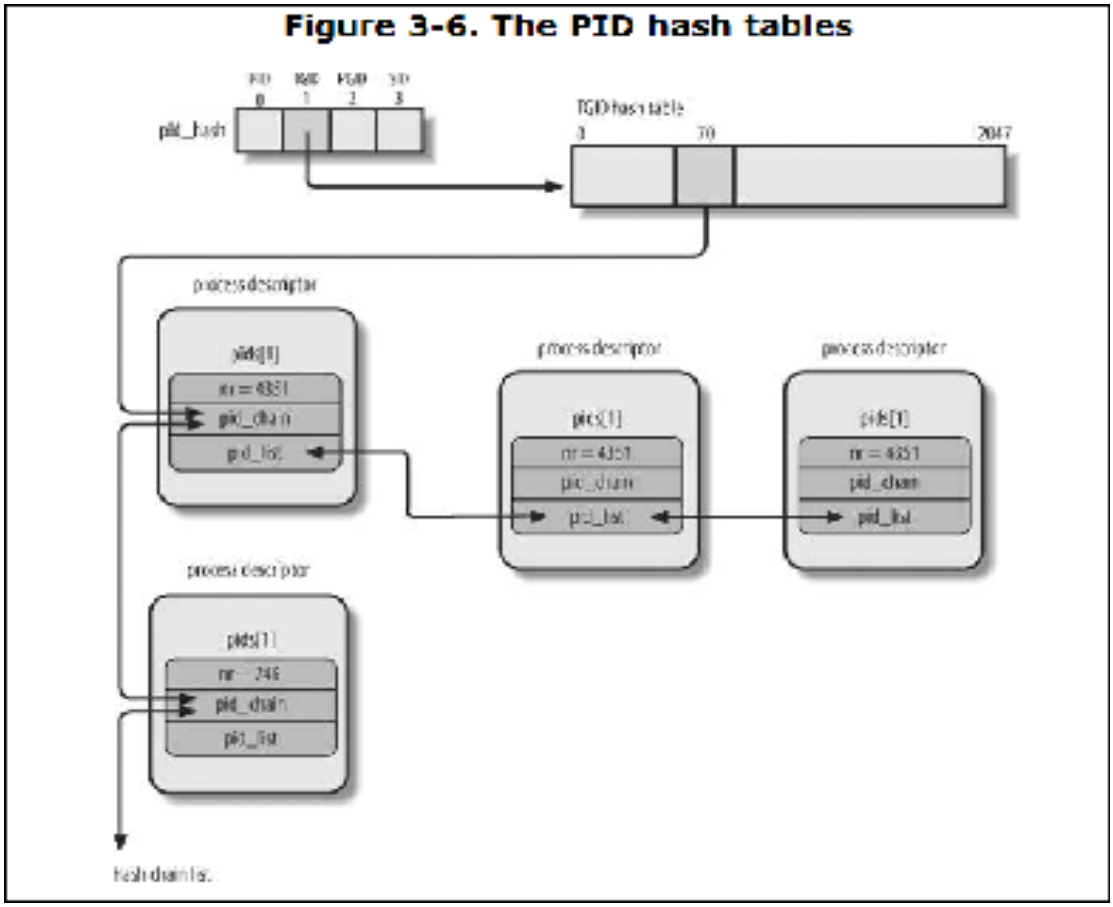
3.2.3.1. The pidhash table and chained lists

四个Hash表。这里使用了《算法导论》中提及的乘法散列

Table 3-5. The four hash tables and corresponding fields in the process descriptor

Hash table type	Field name	Description
PIDTYPE_PID	pid	PID of the process
PIDTYPE_TGID	tgid	PID of thread group leader process
PIDTYPE_PGID	pgrp	PID of the group leader process
PIDTYPE_SID	session	PID of the session leader process

task_struct中内嵌一个struct pid pids[PIDTYPE_MAX]数组，大小为4;
struct pid:
struct pid
{
int nr; //The PID number
struct hlist_node pid_chain;// The links to the next and previous
elements in the hash chain list
struct list_head pid_list;// The head of the per-PID list
};



3.2.4. How Processes Are Organized

runqueue中的进程都是TASK_RUNNING状态。
其他状态:
TASK_STOPPED, EXIT_ZOMBIE, or EXIT_DEAD: 没有用特别的链表组织。
TASK_INTERRUPTIBLE or TASK_UNINTERRUPTIBLE:最常见的是组织在wait queues中。

3.2.4.1. Wait queues

进程在等待某项事件时就会加入等待队列，直到某些事件发生。

2种睡眠进程:
exclusive processes(wait_queue_t-> flags为1): 当事件发生时，选择性的被唤醒。
nonexclusive processes (flags为0) : 当事件发生时,全部被唤醒。

3.2.5. Process Resource Limits

current->signal->rlim域中。

3.3. Process Switch

在进程恢复运行所必须加载进寄存器的数据被称为hardware context，hardware context是process execution context的子集。Linux下，部分hardware context储存在process descriptor中，其它的储存在Kernel Mode stack中。

Linux2.6不使用X86CPU的far jmp指令，far jmp将切换TSS，自动保存旧hardware

context和装载新hardware context。

3.3.2. Task State Segment

虽然Linux不使用far jmp，但仍然要为每个CPU设置TSS。因为：

1. 用户态切换到内核态，需要从TSS中取得内核栈地址。
2. 用户进程访问I/O port需要从TSS中取得I/O Permission Bitmap。

每个CPU的tr寄存器存放了TSSD Selector，tr有两个"影子"寄存器，存放TSSD的Base和Limit域。因此，TSS的读取是很快的。

3.3.2.1. The thread field

每个process descriptor有个thread_struct结构的thread域，内核把要切换出去的进程的hardware context存在这里。

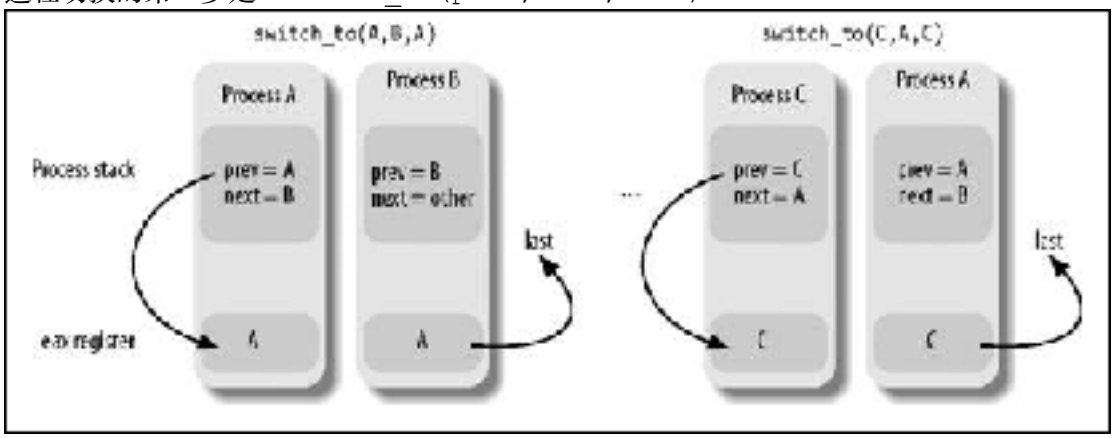
3.3.3. Performing the Process Switch

进程切换两步骤：

1. 切换Page Global Directory，装载新进程地址空间。
2. 切换内核栈和hardware context,这些是进程运行的必要信息。

3.3.3.1. The switch_to macro

进程切换的第二步是：switch_to(prev,next,last)



last变量存放的是上次切换的被替代进程的process descriptor
对切换序列:A->B->C->A
prev:A进程,next:B进程,切换前%eax寄存器存放prev:A的地址。C->切换前，%eax是C的地址，切换后A运行，%eax存的是C地址，它将写入last变量中。

switch_to切换步骤：

- 保存prev,next到%eax,%edx。
- 保存eflags,%ebp到prev的内核栈中。
- %esp存放在prev->thread.esp。
- 装载next->thread.esp到%esp寄存器。从现在开始，开始使用next的内核栈。指令序列真正的从prev切换到了next。因为task_struct是与内核栈联系在一起的。切换内核栈等同于切换了当前进程。
- 保存标记为1代码的地址到prev-> thread.eip。当被切换出去的进程恢复运行时，它将从1处开始运行。
  movl $1f, 480(%eax)

- 在next进程的内核栈中，next->thread.eip压栈。其内容绝大多数情况下是1f的地址。
- 跳转到__switch_to()函数。
- 现在之前被切换出去的进程A又得到了CPU，，此时esp已经指向A的内核栈了。先恢复eflags,%ebp寄存器的值。
- 把%eax内容拷贝到last中。%eax存放的是前一次切换的进程描述符地址。

3.3.3.2. The __switch_to ( ) function

__switch_to()在switch_to中使用jmp汇编跳转的，所以不像call命令那样会把，参数，返回地址压栈。我们必须自己构建。
__attribute__(regparm(3))指定了__switch_to()从寄存器中得到参数。

__switch_to()关键步骤:

1. 得到当前CPU的index。把next_p->thread.esp0装载进TSS的esp0处。当从用户态切换到内核态时，将使用TSS的esp0来提供内核栈地址。
2. 把fs,gs内容存放在prev_p->thread.fs和prev_p->thread.gs。
3. 如果fs,gs在prev_p或next_p中有使用，从next_p中读取fs,gs放入寄存器。
4. 返回prev_p，把prev_p(在函数执行时，从函数外拷入%edi了)从%edi拷贝入%eax。这也是switch_to的last参数的由来。这时，内核栈栈顶存放的是切换前的thread.eip内容，作为返回地址返回,一般是返回到1f处。
   如果next_p进程是刚fork的，之前从未挂起过，那么thread.eip中存的是ret_from_fork()的地址，那么返回地址就是ret_from_fork()。

3.4. Creating Processes

3.4.1. The clone( ), fork( ), and vfork( ) System Calls

轻量级进程使用clone()创建。会创建新的轻量级进程用户栈。
传统的fork()，新进程和父进程指向同一个用户栈，随后发生Copy On Write产生新的用户栈。
vfork(),父子共用一个用户栈，完全的共享，所以父进程必须阻塞，直到子进程退出。

do_fork()主要步骤:

1. 给孩子分配PID。
2. 检测current->ptrace,如果非0，那么父进程在被跟踪，do_fork()检查调试器是否要跟踪孩子。如果孩子不是内核线程，那么设置CLONE_PTRACE。
3. 调用copy_process()拷贝出一个process descriptor。这是do_fork()的主要工作。
4. 如果CLONE_STOPPED被设置或孩子要被跟踪（p->ptrace为1），那么把孩子的状态设为TASK_STOPPED，并给它一个信号SIGSTOP。孩子将保持TASK_STOPPED状态直到另一个进程发送SIGCONT信号给它，使之变成TASK_RUNNING状态。
5. 如果CLONE_STOPPED未被设置，则调用wake_up_new_task( )函数。
   a) 调整父进程和子进程的调度参数。
   b) 如果孩子和父亲在同一个CPU上运行（当内核fork新进程时，父进程有可能被移动到另一个CPU），并且没有共享同一个page tables。那么强迫在runqueue中，孩子排在父亲前面，让孩子先调度，这样就可以减少不必要的COW。

c) 否则孩子和父亲不在同一个CPU上或它们共享page tables，那么把孩子插在父亲之前所在的runqueue队列的末尾。

6. 如果CLONE_STOPPED设置，进入TASK_STOPPED状态。
7. 如果父进程正在被跟踪，它把孩子的PID放入current->ptrace_message，调用ptrace_notify( )(暂停当前进程，发送SIGCHLD给它父亲)。孩子的"祖父"就是调试器。SIGCHLD通知调试器，currunt进程fork出了一个孩子，PID号在current->ptrace_message中。
8. 如果是CLONE_VFORK，把父进程插入孩子的等待队列并挂起父进程，直到孩子退出。
9. 返回孩子的PID。

3.4.1.2. The copy_process( ) function

新创建的进程，其thread.eip存放的是ret_from_fork()地址，因此新进程第一次被调用时，会首先运行ret_from_fork()。
ret_from_fork()将使子进程从fork()系统调用后一条指令处开始执行，对子进程返回0，对父进程返回子进程的PID。

copy_process()步骤:

1. 检测flags合法性。
2. 调用dup_task_struct()获取孩子的process descriptor。步骤为:
   a. tsk = alloc_task_struct( );
   b. ti = alloc_thread_info();
   c. 复制父亲的task_struct和thread_info到孩子的相应结构。
   d. 返回tsk;
3. 资源限制检测。
4. tsk->pid置为新进程PID。
5. 调用copy_thread()初始化孩子的内核栈，孩子的thread.esp初始化为内核栈的基地址。Thread.eip初始化为ret_from_fork( )地址。
6. 初始化tsk->exit_signal为clone_flags参数的低bit，CLONE_THREAD置位，则tsk->exit_signal初始化为-1。
7. 调用sched_fork()完成跟调度有关的数据结构的初始化。把进程状态置为TASK_RUNNING，preempt_count置1（关闭抢占），与父进程平分时间片。
8. 初始化父亲关系。如果CLONE_PARENT或CLONE_THREAD设置了，tsk->real_parent 和 tsk->parent置为current->real_parent;否则置为current。可见，同一个线程组中的线程父亲都是同一个进程（第一个线程的父亲）。
   注：线程退出时，real_parent会被置为线程组中的其它线程。如果没有线程了，就置为init
9. 如果孩子是一个线程组leader(CLONE_THREAD为0)
   tsk->tgid = tsk->pid;
   tsk->group_leader = tsk.
   插入PIDTYPE_TGID, PIDTYPE_PGID, PIDTYPE_PID and PIDTYPE_SID队列.
10. 如果孩子不是一个线程组leader(CLONE_THREAD为1)
    tsk->tgid = current->tgid。
    tsk->group_leader = current->group_leader。
    插入PIDTYPE_PID 和 PIDTYPE_TGID队列。（以及leader进程的内部队列）

11. 收尾。略

3.4.2. Kernel Threads

kernel_thread(int (*fn)(void *), void *arg, unsigned long flags)
创建一个内核线程。它接收一个内核函数地址，内核函数参数，flags。本质上是调用
do_fork(flags|CLONE_VM|CLONE_UNTRACED, 0, pregs, 0, NULL, NULL);
pregs参数等于新线程的内核栈地址。kernel_thread()在内核栈中创建基本数据。
ebx存放fn，edx存放arg。
eip存放如下汇编代码片段的地址：

```
movl %edx,%eax
pushl %edx
call *%ebx
pushl %eax
call do_exit
```

可见，内核线程执行fn(arg)，当函数执行完毕后，把返回值传递给_exit系统调用。

3.4.2.2. Process 0

0号进程使用静态的数据结构。

1. process descriptor在init_task中
2. thread_info在init_thread_union中
3. init_mm
4. init_fs
5. init_files
6. init_signals
7. init_sighand
8. master kernel Page Global Directory在swapper_pg_dir中。

再多处理器系统中，对每一个CPU都有一个0号进程。刚开机时，只有一个CPU运行，0号CPU上的swapper process初始化内核数据结构，然后启用其它CPU。创建额外的swapper process进程，进程号0。

3.4.2.3. Process 1

1号进程是常规进程，一直运行到关机。监视和创建其它所有进程。

3.5. Destroying Processes

3.5.1.1. The do_group_exit( ) function

对线程组里的线程（不等于current）发信号SIGKILL。其他线程收到SIGKILL后会调用do_exit退出。current进程do_exit。

3.5.1.2. The do_exit( ) function

只有当线程组的最后一个线程退出才会发信号给父进程，进入EXIT_ZOMBIE状态。
如果线程组中还有线程，则把退出线程的孩子成为线程组中另一线程的孩子。退出线程直接死掉，进入EXIT_DEAD状态。回收资源。

Chapter 4. Interrupts and Exceptions

同步中断：异常，CPU产生。
异步中断：中断，外设产生。

4.2. Interrupts and Exceptions

每个中断或异常被一个0~255的数字代替。Intel使用一个8bit的vector。不可屏蔽的中断或异常固定死了，可屏蔽的中断能被programming the Interrupt Controller选择使用。

4.2.1. IRQs and Interrupts

每个硬件通过Interrupt ReQuest (IRQ) line连接到Programmable Interrupt Controller。

PIC的工作步骤:

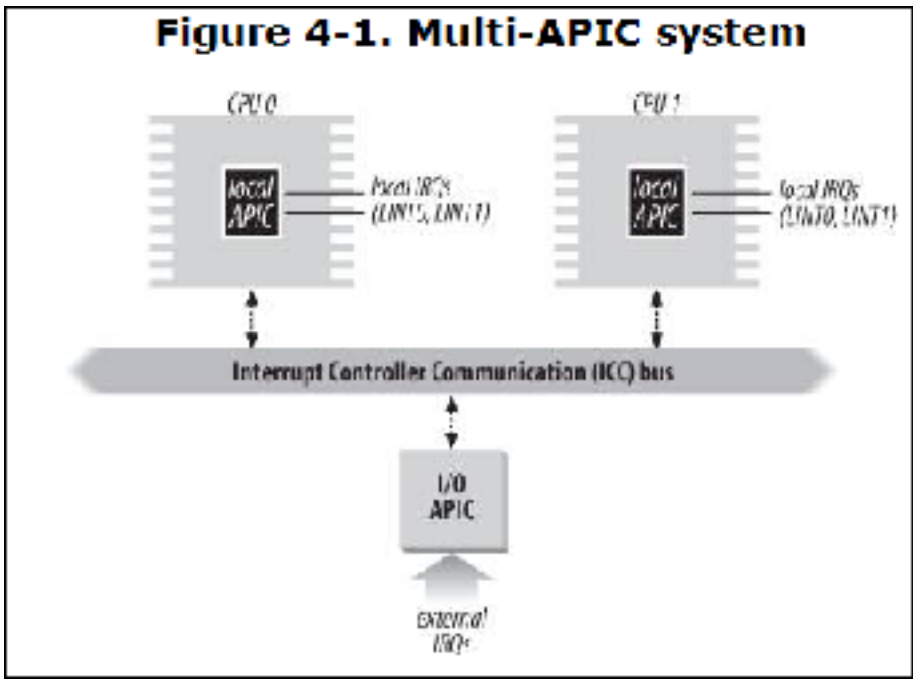
1. 监视IRQ lines，如果有多个信号，选择低位引脚的信号。
2. 如果有信号发生:
   把信号转换成vector。
   把vector存储在Interrupt Controller I/O port，CPU通过数据总线读取它。
   发送一个信号给处理器的INTR引脚。
   等待CPU确认中断信号，确认是通过CPU向Interrupt Controller I/O port写数据。确认完成后，清除INTR。
3. 回到1步。

Intel的缺省设置是：对应vector与IRQ n+32联系。对应关系可以修改。

每个IRQ线能被选择性的禁止，当恢复IRQ时，中断信号马上能发给CPU,这在同种中断的串行处理中使用。

当CPU的eflags->IF为0时，所有可屏蔽的中断都被CPU忽略。cli和sti分别用来清除和置位。

4.2.1.1. The Advanced Programmable Interrupt Controller (APIC)



4.2.3. Interrupt Descriptor Table

Interrupt Descriptor Table (IDT)把每个中断或异常vector与处理函数的地址相联系。IDT表必须在启动中断前时初始化。

idtr CPU寄存器允许IDT存在内存的任何位置。idtr指定了IDT的基地址和长度。

IDT中的三种表项(gate descriptor):

1. Task gate:当中断发生时，包含了要替换当前进程的新进程的TSS selector（Linux很少用了，因为没有TSS切换）。
2. Interrupt gate:包含了中断或异常处理函数地址的Segment Selector和offset。当控制到达处理函数前，IF flag清0,关闭中断。
3. Trap gate:类似与中断门，但不关中断。

4.2.4. Hardware Handling of Interrupts and Exceptions

中断或异常处理步骤:

1. 确定vector i。
2. 从idtr 指定的IDT读取第i项(假设是中断门或陷阱门)。
3. 从gdtr指定的GDT中读取Segment Description，它指定了处理函数所在的段。
4. 检测中断源的合法性。比较cs中的CPL和GDT中Segment Description的DPL。如果CPL数值上小于DPL，将产生"General protection"异常，因为中断处理函数的特权级不可能小于产生中断时的程序特权级。对于异常，还要进一步检测，比较CPL和gate descriptor的DPL。如果DPL数值上小于CPL，将产生"General protection"异常。这一步检测阻止了用户进程对特定中断门和陷阱门的访问。
5. 检测是否有特权级的变化，如果CPL不等于Segment Description的DPL，说明特权级变化了，那必须使用不同的栈:
   a. 读tr ,访问 TSS段。
   b. 在TSS中找到合适的ss,esp值，装载到ss和esp寄存器。
   c. 在新的栈中，把之前的ss,esp存在栈中。
6. 保存eflags,cs,eip的内容到栈中。

7. 如果异常产生了错误码，保存在栈中。
8. 把GDT中指定的段和偏移装载入cs和eip寄存器。那么，处理函数将接下来执行了。

中断或异常处理完了后，调用iret指令，步骤：
1. 把保存在栈里的cs,eip,eflags值装载进对应寄存器。如果有错误码，那应该在iret前弹出。
2. 检测处理函数的CPL是否等于cs中的CPL（判断是否发生了栈切换）。如果相等，iret中止。否则：
3. 从栈中读取之前特权级使用的ss，esp，装载进ss，esp寄存器。
4. 检测ds，es，fs，gs段寄存器，看是否有DPL数值上小于CPL。如果是，清空对应的寄存器。

4.3. Nested Execution of Exception and Interrupt Handlers

中断和异常处理函数可以嵌套，
中断嵌套的代价是中断处理函数不允许阻塞，在中断处理时除非有中断发生，否则不会有进程切换发生。
"页面错误"不会导致更多的异常，因此最多有2个内核控制路径存在栈中（一个是系统调用一个是页面错误）。
中断并不像异常那样绑定在一个进程上。
中断处理函数能抢占其它中断处理或异常处理函数。相反的，一个异常处理函数不会抢占中断处理函数。在内核中唯一的异常是"页面错误"，但中断处理不会产生"页面错误"，也就不会引发进程切换。

4.4. Initializing the Interrupt Descriptor Table

int指令能够产生一个异常，vector号能从0-255。所以，必须仔细的初始化IDT，正常的系统调用，DPL设为3。特权级不符将导致"General protection"异常。

4.4.1 Interrupt, Trap, and System Gates

Linux对interrupt descriptors稍有不同的解释：

- Interrupt gate:Intel中断门，所有的Linux中断使用中断门，不允许用户进程穿越。
- System gate: Intel陷阱门，能被用户进程穿越。3种用户异常对应vector 4, 5, 128，对应into,bound,int 0X80被System gate初始化。
- System Interrupt gate:Intel中断门，能被用户进程穿越。唯一的vector 3对应用户态指令int3。
- Trap gate:Intel陷阱门，绝大多数Linux异常使用陷阱门，不能被用户进程穿越。
- Task gate:Intel任务门，不能被用户进程穿越，Linux用来处理"Double fault "异常。

4.5. Exception Handling

异常处理标准步骤：
1. 保存寄存器内容到内核栈中。
2. 利用C函数处理异常。
3. 调用ret_form_exception()退出异常。

4.5.1. Saving the Registers for the Exception Handler

用handler_name代替特定的异常处理函数。每个异常处理函数以下面的汇编开始：

```
handler_name:
pushl $0 /* only for some exceptions */
pushl $do_handler_name
jmp error_code
```

当异常发生时，如果不会自动插入错误码，则要用一条指令压入一个0做占位符。然后异常处理函数压入堆栈。

error_code执行下列步骤：

1. 保存处理函数要用的寄存器值到栈中。
2. 把出错码保存到%edx中，出错码位置用-1代替。这个值用来区分系统调用和其它异常。
3. 把当前内核栈顶地址存放在%eax中，这个地址就是第一步中保存的最后一个寄存器内容地址。
4. 装载用户数据Segment Selector到ds和es寄存器。
5. 调用异常处理函数，其地址保存在%edi中。

异常处理函数两个参数，一个从%eax读，一个从%edx读。

4.6. Interrupt Handling

大多数异常处理只是简单的发送一个信号给产生异常的进程，真正的异常的处理被推迟到收到信号时，因此处理异常很快。

4.6.1. I/O Interrupt Handling

IRQ sharing:中断处理函数执行多个中断服务程序（ISRs），每个ISR对应一个共享IRQ线的设备。因为不可能知道哪个设备发送了中断，每个ISR都会执行，判断它对应的设备需要处理。如果发现了，则该ISR处理该中断。

IRQ dynamic allocation: 只到最后时刻，一根IRQ线才与特定设备联系在一起。

不是中断处理中所有动作都是同样紧急的。运行时间长的非关键操作应该被推迟。因为当中断处理时，同种中断是被忽略的。最重要的是，如果中断有阻塞操作，如I/O操作，整个系统就都将冻住。

Linux把中断处理动作分为三类：

Critical: 例如对PIC的中断确认，PIC重编程，设备控制，更新被设备和处理器访问的数据结构。这些操作能被很快执行，而且很关键。因为它们必须尽可能快的执行完。此过程中，可屏蔽的中断被禁止。

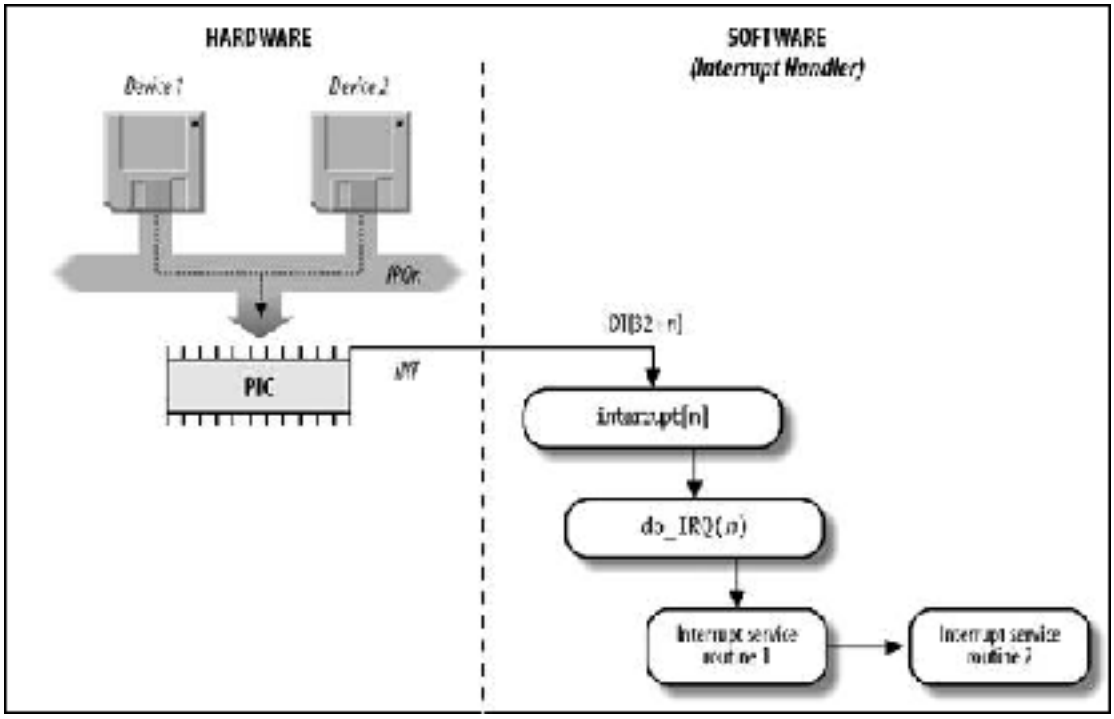
Noncritical: 例如更新只被处理器访问的数据结构。这些动作完成很快，在开中断条件下。

Noncritical deferrable: 例如把buffer内容拷贝到进程空间。这类操作会被延迟进行。

I/O中断处理基本步骤：

1. 保存IRQ值和寄存器内容到内核栈。
2. 发送确认给PIC。
3. 执行中断处理函数。
4. 通过ret_from_intr返回。

4.6.1.1. Interrupt vectors



(*注:由中断向量确定了interrupt[n]后,do_IRQ(n)宏会把n扩展成对应处理函数)

```
void __init init_IRQ (void)
{
……
for (i = 0; i < (NR_VECTORS - FIRST_EXTERNAL_VECTOR); i++) {
       int vector = FIRST_EXTERNAL_VECTOR + i;
       if (i >= NR_IRQS)
           break;
       if (vector != SYSCALL_VECTOR)
           set_intr_gate(vector, interrupt[i]);
    }
……
}
```

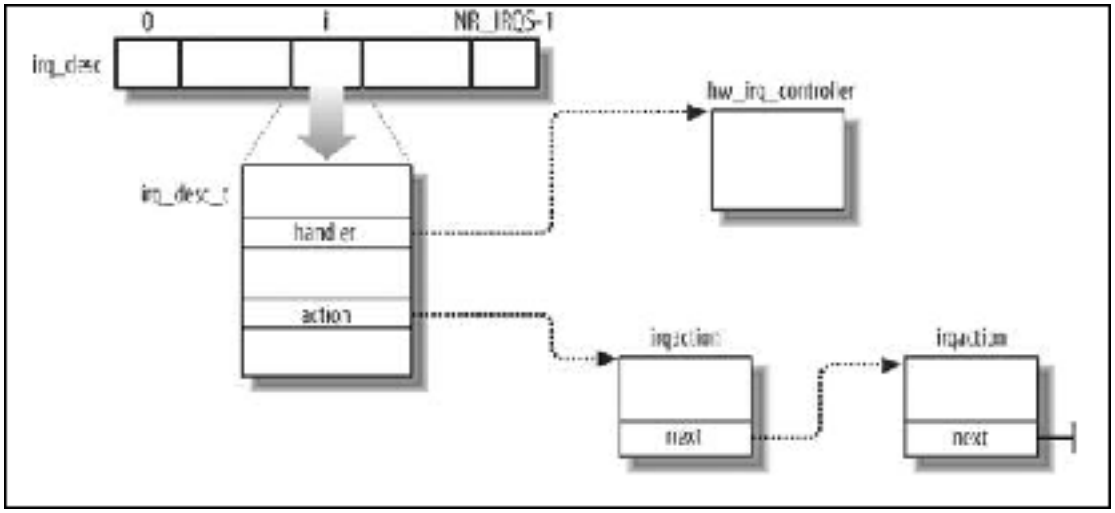
interrupt[]数组存放:

```
#define IRQ(x,y) \
        IRQ##x##y##_interrupt

#define IRQLIST_16(x) \
         IRQ(x,0), IRQ(x,1), IRQ(x,2), IRQ(x,3), \
         IRQ(x,4), IRQ(x,5), IRQ(x,6), IRQ(x,7), \
         IRQ(x,8), IRQ(x,9), IRQ(x,a), IRQ(x,b), \
         IRQ(x,c), IRQ(x,d), IRQ(x,e), IRQ(x,f)

void (*interrupt[NR_IRQS])(void) = {
         IRQLIST_16(0x0)
};
```

4.6.1.2. IRQ data structures



每一个中断向量有自己的irq_desc_t描述符。所有描述符组成了irq_desc数组。
irq_desc_t关键成员说明：
1. handler:指向PIC对象。
2. handler_data:PIC方法使用的数据。
3. action:一个irqaction描述符队列。
4. status:描述IRQ线状态。
5. depth:0该IRQ线开放，1该IRQ线关闭。

因为可能有多个设备共享一根IRQ线，所以irqaction结构链成了队列。每个irqaction对应一个ISR。
irqaction结构关键成员：

- handler:对应一个I/O设备的ISR程序。
- flags:描述IRQ线和I/O设备的关系。
- name:I/O设备名字。
- dev_id:I/O设备的私有数据，典型的，可以是I/O设备的设备号，或指向设备驱动的数据。
- next:指向下一个irqaction。

4.6.1.3. IRQ distribution in multiprocessor systems

Linux的SMP结构，意味着内核对所有CPU公平的看待。那么，对IRQ信号以轮询的方式在各CPU中分配。每一个CPU大约花费了相等比例的时间来服务I/O中断。IRQ的公平分配是由muti-APIC系统来硬件实现的。

在某些情况下，硬件对IRQ的公平分配会失效，所以,Linux内核利用了一个内核线程:kirqd来纠正它。kirqd周期性的调用do_irq_balance()，跟踪每个CPU在时间间隔内的中断次数，如果发现某个CPU负担过重，就把某IQR移动到另一个CPU。

kirqd内核线程实现了一种新特性—the IRQ affinity of a CPU（通过修改Interrupt Redirection Table entries of the I/O APIC）,就能把IRQ请求定向到特定CPU。

4.6.1.4. Multiple Kernel Mode stacks

根据内核编译选项，可以使用4KB大小的thread_union结构，那么将使用多个内核栈：
1. exception stack: 处理异常，这个栈页面包含在每个进程的thread_union结构

中。因此，每个进程有不同的exception stack。

2. hard IRQ stack:处理中断的内核栈，每个CPU都有对应一个hard IRQ stack,，每个栈占一个页面。
3. soft IRQ stack: 处理deferrable functions,每个CPU都有一个soft IRQ stack，每个栈占1个页面。

所有的hard IRQ stacks包含在hardirq_stack数组中，所有的soft IRQ stacks包含在softirq_stack数组中。每个栈底存放thread_info结构。其它内存做为栈使用。

4.6.1.5. Saving the registers for the interrupt handler

每个interrupt数组成员都是一个中断处理函数，执行汇编：

```
pushl $n-256
jmp common_interrupt
```

保存IRQ号-256。内核通过负数来区分系统调用，然后跳转到common_interrupt处。

```
common_interrupt:
  SAVE_ALL
  movl %esp,%eax
  call do_IRQ
  jmp ret_from_intr
```

SAVE_ALL保存了所有中断处理需要使用的CPU寄存器,除了eflags,ce,eip,ss,esp（它们已经自动保存了）。SAVE_ALL宏接着装载用户数据的selector(__USER_DS)到ds和es。

保存完寄存器后，调用do_IRQ()，然后ret_from_intr()。

4.6.1.6. The do_IRQ( ) function

do_IRQ()函数执行下列动作：

1. irq_enter()宏，增加thread_info->preempt_count(这个值被划分成了几个域，这里是增加的代表中断嵌套的计数器)
2. 如果thread_union是4KB，就切换到hard IRQ stack:

a.得到thread_info地址(利用当前esp值得到)。

b.比较a处得到的地址与该CPU的hard IRQ stack地址。如果相等，说明内核已经在使用该hard IRQ stack，跳转到3步。这种情况发生在当内核在处理某中断时，又发生了一个中断。

c. 两地址不相等，这里就要发生内核栈的切换。把current process descriptor存放到irq_ctx中的thread_info->task。这样，current宏仍然可以得到当前process descriptor。

d. 把esp寄存器当前值保存在irq_ctx中的thread_info->previous_esp（内核oops时调试用）。

e.把irq_ctx中的内核栈地址装载进%esp(irq_ctx地址+4096,注意，这个栈是空的，之前没有嵌套中断)，之前%esp值保存在了%ebx寄存器中。

3. 调用__do_IRQ()函数，参数regs，IRQ号保存在regs->orig_eax中。
4. 如果2e的栈切换完成，函数恢复原来使用的栈，利用把%ebx拷贝到%esp。这样又切换回了原来那个栈。
5. 执行irq_exit（）宏，减少之前增加的中断嵌套计数。检查是否有deferrable kernel functions正在等待执行。
6. 结束，跳转到ret_from_intr()。

4.6.1.7. The __do_IRQ( ) function

__do_IRQ()函数接收IRQ号(%eax中)，pt_regs结构（%edx中）。

函数代码等同于下列片段:

```
spin_lock(&(irq_desc[irq].lock));
irq_desc[irq].handler->ack(irq);
irq_desc[irq].status &= ~(IRQ_REPLAY | IRQ_WAITING);
irq_desc[irq].status |= IRQ_PENDING;
if (!(irq_desc[irq].status & (IRQ_DISABLED | IRQ_INPROGRESS))
    && irq_desc[irq].action) {
    irq_desc[irq].status |= IRQ_INPROGRESS;
    do {
            irq_desc[irq].status &= ~IRQ_PENDING;
            spin_unlock(&(irq_desc[irq].lock));
            handle_IRQ_event(irq, regs, irq_desc[irq].action);
            spin_lock(&(irq_desc[irq].lock));
        } while (irq_desc[irq].status & IRQ_PENDING);
    rq_desc[irq].status &= ~IRQ_INPROGRESS;
    }
irq_desc[irq].handler->end(irq);
spin_unlock(&(irq_desc[irq].lock));
```

在访问IRQ descriptor前，内核请求一个spin lock。因为同一种中断有可能在不同CPU上产生，为了确保只有一个进程访问IRQ descriptor，所以使用了spin lock。

得到spin lock后。函数确认IRQ给PIC，并在CPU上屏蔽该IRQ。这样就串行了同种IRQ。注意到，在CPU穿越中断门时，CPU关闭整个中断。

在I/O APIC中，情况要复杂很多。Local APIC确保在中断处理时，同种IRQ不会被它接收，但同种中断可以被不同CPU接收。

在真正执行中断处理函数前，__do_IRQ()函数检测如下标志位:

- IRQ_DISABLED设置: CPU在IRQ线被禁止时也能中断处理，IRQ_DISABLED将会在"丢失的中断恢复"中利用到。__do_IRQ()函数会确认该中断，但不会执行处理程序。
- IRQ_INPROGRESS设置: 在SMP中，另一个CPU可能会收到一个正在处理的中断。第二个CPU并不是推迟对该中断的处理，Linux使第二个CPU立即返回。这种处理方式，让内核结构简单，中断不需要重入，而是串行了。而且，立即返回的CPU不会污染它的硬件cache。
- irq_desc[irq].action is NULL: 仅发生在内核探测硬件设备过程中。

如果决定要执行中断处理，IRQ_INPROGRESS置位。每次迭代中会清除IRQ_PENDING。当执行完中断处理后，会再次检测是否有IRQ_INPROGRESS置位（由其它CPU 引起），是的话，再次执行中断处理函数。

4.6.1.8. Reviving a lost interrupt

假设某CPU可以接受某中断（PIC可以发生该中端），然而该中断被另一个CPU置为IRQ_DISABLED，因此，该CPU仅仅置位IRQ_PENDING。然后就返回了。

为了解决这个问题。enable_irq()函数，用来启用IRQ线（修改PIC）。检测是否有中断丢失，如果是。函数强迫硬件产生一个新的该中断。

4.6.1.9. Interrupt service routines

ISR处理步骤:
1. 打开中断，如果SA_INTERRUPT清零。
2. 调用响应函数。
3. 关闭中断。
4. 结束。

(个人感觉，在ISR开始前，ISR结束后都是关中断的，未验证)

4.7. Softirqs and Tasklets

同种中断是被串行的，然而对deferrable task却是在开中断环境下执行的。Softirq和tasklet是紧密相连的，tasklet是在softirq之上实现的。

中断上下文：指内核在中断处理或deferrable task中执行。

Softirq是在编译时静态分配的，tasklet能在运行时初始化（例如加载模块时）。Softirq能同时在不同CPU上执行，甚至是同种类型的softirq。因此，sotfirq是种可以重入的函数，必须用spin lock保护它的数据结构。Tasklet同种类型被串行化，同种类型的tasklet不能在不同CPU上同时执行。串行化tasklet使驱动开发简化了，因为它不能重入。

大体来说，4种关于deferrable function的操作:
1. Initialization:定义一个deferrable function，这个操作通常在内核初始化或模块加载时进行。
2. Activation: 标记deferrable function，使之它在下一次调度时执行。Activation能在任何时候进行（甚至在中断处理时）。
3. Masking: 选择性的屏蔽deferrable function，即使被激活了也不能执行。
4. Execution: 执行所有pending deferrable function。

Activation和Execution肯定在同一个CPU上进行，可以更好的利用cache，不过，把函数绑定在某CPU上可能会造成负载失衡。

4.7.1. Softirqs

一共有6种softirq(优先级由高到低):
HI_SOFTIRQ,TIMER_SOFTIRQ,NET_TX_SOFTIRQ
NET_RX_SOFTIRQ,SCSI_SOFTIRQ,TASKLET_SOFTIRQ

4.7.1.1. Data structures used for softirqs

主要结构:
```
struct softirq_action
{
        void    (*action)(struct softirq_action *);
        void    *data;
};
```
存放在softirq_vec[32]数组中，只有前6项使用。数组index代表了优先级。
thread_info->preempt_count存放了重要的数据,用来追踪内核抢占和嵌套:
1. 0-7:抢占计数器(<=255)，记录了CPU显式禁止内核抢占的次数；如果为0，代表内核抢占从未被显式禁止过。
2. 8-15:软中断嵌套计数器(<=255);每次进入softirq，都会使之加一，它记录了一共有多少个softirq在执行。(不同CPU中执行，softirq是不能嵌套的)。

3. 16-27: 硬中断嵌套计数器（<=4096）；记录了嵌套的中断数目(如果只有一个中断在处理中，该值为1)。
4. 28-31: PREEMPT_ACTIVE flag;

preempt_count的作用:抢占式内核既可以被显式的关闭抢占（0-7bit），或者在中断上下文中。只要preempt_count非零，抢占即被关闭。

in_interrupt()宏检测preempt_count中的硬中断和软中断计数，只要一个非0，就返回1。

4KB内核栈：如果栈实在irq_ctx，那么肯定是在中断上下文中。

Softirq pending mask，存放在irq_cpustat_t->__softirq_pending。

4.7.1.2. Handling softirqs

raise_softirq( )来activate softirq，接收参数nx:

1. 调用local_irq_save保存IF和eflags并关中断。
2. 标记softirq bit mask(irq_cpustat_t->__softirq_pending)中的第nx位。
3. 如果in_interrupt()返回1，跳转到5。这种情况发生指出raise_softirq()要么在中断向下文中，要么软中断被关闭了。
4. 否则，调用wakeup_softirqd()唤醒ksoftirqd kernel thread of the local CPU。
5. local_irq_restore恢复IF和eflags.

周期性检测是否有softirq pending。在内核某些点处也会检测：

1. 当内核调用local_bh_enable()启用softirq时(bh名字是指"bottom half"，但"bottom half"在2.6内核里已经不存在了)。
2. 当do_IRQ()完成I/O中断，调用irq_exit()宏时。
3. 当使用I/O APIC。smp_apic_timer_interrupt( )函数完成处理local timer interrupt时。
4. 当ksoftirqd内核线程被唤醒时。

(各个检测点，都要检测是否在中断上下文中，如果是，不进行softirq处理)。

4.7.1.3. The do_softirq( ) function

local_softirq_pending()非0说明有softirq pending。接着执行:

1. 如果in_interrupt()返回1，说明在中断上下文中或softirq被关闭了，函数返回。
2. 执行local_irq_save关中断。
3. 如果thread_union是4KB大小，切换内核栈到soft IRQ stack,过程和do_IRQ()中相似，不过使用的是hardirq_ctx。
4. 调用__do_softirq()函数。
5. 如果切换了内核栈，现在要切换回来。
6. 执行local_irq_restore开中断。

4.7.1.4. The __do_softirq( ) function

__do_softirq()函数读取softirq bit mask 执行对应的deferrable function。当在执行一个softirq 函数时，新的softirq pending可能会产生，为了降低延时，因此，__do_softirq()只执行固定次数的迭代。剩下未处理的pending softirq将被ksoftirqd内核线程处理。__do_softirq()执行步骤是:

1. 初始化迭代计数器为10。

2. 拷贝softirq bit mask of local CPU到pending变量中。
3. 调用local_bh_disable()增加softirq计数器(preempt_count中)。关闭了deferrable function。因为softirq可以被中断，而中断又会产生softirq处理检测，必须把softirq串行起来，所以要关掉deferrable function。
4. 清除softirq bitmap of the local CPU（它的值已在步骤2保存下来了），这样新的softirq能被activate。
5. local_irq_enable()开中断。
6. 对每个softirq调用处理函数。
7. local_irq_disable()关中断。
8. 拷贝softirq bit mask到pending变量，减少迭代计数器。
9. 如果pending非0，说明又有softirq被activate。跳转到4。
10. 迭代器到0了，如果还有softirq要处理，调用wakeup_softirq()唤醒内核线程来处理softirq。
11. 从softirq countr(preempt_count中)中减一，使deferrable function变成可重入。

对于local_bh_enable()(假设之前是关deferrable function的):
1. local_bh_enable()首先执行sub_preempt_count(SOFTIRQ_OFFSET - 1)，硬中断计数器不受影响。软中断计数器变成0。抢占计数器是1。
2. 如果没有在中断上下文中，就执行do_softirq。此时是关抢占的。
3. 把抢占计数器减一，此时可以抢占了。

对于local_bh_disable():
1. 执行add_preempt_count(SOFTIRQ_OFFSET)，软中断计数器加一，表明进入了softirq，也就关闭了抢占。local_bh_disable()多次调用会不断增加软中断计数器。

对于in_interrupt():
1. 它检测preempt_count硬中断计数器和软中断计数器。是否有一个非0。硬中断计数器非0说明正在处理硬中断中。软中断计数器非0说明，被关软中断了，其实也是在软中断处理环境中。

4.7.1.5. The ksoftirqd kernel threads

每个CPU有自己的ksoftirqd内核线程。每当被唤醒，就检测是否有softirq要处理，如果有，就关闭抢占，执行do_softirq，完成后接着睡眠。

ksoftirqd是一种折中方案。当softirq产生过于频繁时。如果是在do_softirq中忽略新产生的softirq,会造成大延时。如果不停的处理新softirq，会造成用户进程得不到相应。

小结:
无论是硬中断还是软中断，在真正执行处理函数时，都在进行外围准备工作，此时是关闭所有中断的，直到真正进入了处理程序，才开放中断。

4.7.2. Tasklets

Tasklets是一种在I/O驱动的deferrable function中使用很好的机制。Tasklet建立在两种sofirq: HI_SOFTIRQ 和 TASKLET_SOFTIRQ之上。这两种softirq只有优先级的不同，其它都一样。

Tasklets和高优先级的tasklets分别存放在tasklet_vec和tasklet_hi_vec数组中。数组大小为NR_CPUS，类型为tasklet_head，这个结构指向一个tasklet descriptors链表。tasklet descriptors对应的结构是：tasklet_struct。

```
struct tasklet_struct
 {
    struct tasklet_struct *next;
    unsigned long state;
    atomic_t count;
    void (*func)(unsigned long);
    unsigned long data;
};
```

State有两种标志：
TASKLET_STATE_SCHED：如果设置了，说明tasklet正在pending，也说明tasklet被插入tasklet_vec或tasklet_hi_vec数组；
TASKLET_STATE_RUN：说明tasklet正在被执行，用于使同种tasklet不能在多个CPU中同时执行。

Tasklet是自己分配的，再用过tasklet_init把tasklet_struct,处理函数，可选数据组合在一起。

tasklet_disable(tasklet_struct *t)可以把t关闭，这样就不会发生t的重入操作。

为了activate tasklet，我们可以使用tasklet_schedule()函数或tasklet_hi_schedule()函数,它们很相似：

1. 检查TASKLET_STATE_SCHED，如果是，说明tasklet已经调度好了，返回。
2. 执行local_irq_save关中断。
3. 把tasklet descriptor插入tasklet_vec[n]或tasklet_hi_vec[n]表头。
4. 调用raise_softirq_irqoff() activate HI_SOFTIRQ 或 TASKLET_SOFTIRQ softirq。
5. 执行local_irq_restore开中断。

接着是tasklet的执行。tasklet_hi_action或tasklet_action，他们很相似：

1. 关中断。
2. 得到CPU号。
3. 把tasklet_vec[n]或tasklet_hi_vec[n]存的结构的地址放在list局部变量中。
4. tasklet_vec[n]或tasklet_hi_vec[n]置为NULL。
5. 开中断。
6. 对list链上的每一个tasklet：
   a.SMP系统中,检测TASKET_STATE_RUN标志。
   如果它被设置，说明有同种类型的tasklet在另一个CPU上执行，因此，函数将把这个tasklet重新插入数组，再次activate TASKLET_SOFTIRQ 或 HI_SOFTIRQ。在这种情况下，tasklet被推迟到了其它CPU上没有同种tasklet执行的时候。
   b.检测tasklet的count位，判断它是否被关闭了。如果被关闭了，就清除TASKET_STATE_RUN标志,重新插入数组，在次activate softirq。

C.如果tasklet能执行，那么清除TASKLET_STATE_SCHED,执行tasklet函数。注意到，除非tasklet函数activate自己，否则每一个tasklet最多执行一次tasklet函数。

4.8. Work Queues

deferrable function和work queues区别很大，deferrable function在中断上下文中执行，而work queues在进程上下文中执行。那么work queues执行函数可以被阻塞。因为在中断上下文中，不会有切换发生。deferrable function和work queues都不能访问访问用户地址空间。因为deferrable function更不不能确定是哪个用户进程在执行，而work queues则在内核线程中执行，完全没有用户空间可以访问。

4.8.1.1. Work queue data structures

Work queue的主要数据结构是：workqueue_struct

```
struct workqueue_struct {
       struct cpu_workqueue_struct cpu_wq[NR_CPUS];
       const char *name;
       struct list_head list;  /* Empty if single thread */
};

struct cpu_workqueue_struct {
       spinlock_t lock;
       long remove_sequence;   /* Least-recently added (next to
run) */
       long insert_sequence;   /* Next to add */
       struct list_head worklist;/* Head of the list of pending
                                  functions*/
/* Wait queue where the worker thread waiting for more work to be
done sleeps*/
       wait_queue_head_t more_work;
/* Wait queue where the processes waiting for the work queue to
be flushed sleep*/
      wait_queue_head_t work_done;
       struct workqueue_struct *wq;
/*Process descriptor pointer of the worker thread of the
structure
*/
       task_t *thread;
       int run_depth;          /* Detect run_workqueue()
recursion depth */
};
```

4.8.1.2. Work queue functions

每一个work queue不断的在worker_thread（）中执行一个循环，大部分时间，这个线程在睡眠中，等待一些work入队。一旦被唤醒，work线程调用run_workqueue()函数，这个函数将移除work_struct descriptor，然后执行相应的pending函数。

4.9. Returning from Interrupts and Exceptions

thread_info->flags，最重要的标志是：

1. TIF_SIGPENDING:有pending信号。
2. TIF_NEED_RESCHED:必须调度。

两个返回调用点：

```
ret_from_intr()
ret_from_exception()
```

4.9.1.1. The entry points

两个返回调用,等同于下列代码:

```
ret_from_exception:
    cli ; missing if kernel preemption is not supported
ret_from_intr:
    movl $-8192, %ebp ; -4096 if multiple Kernel Mode stacks are
used
    andl %esp, %ebp
    movl 0x30(%esp), %eax
    movb 0x2c(%esp), %al
    testl $0x00020003, %eax
    jnz resume_userspace
    jpm resume_kernel
```

因为中断返回时，中断是被关闭的。所以只有异常返回需要关中断（防止发生中断，然后抢占发生）。

接着，同过保存在栈中的cs和eflags值，判断被中断的进程是用户空间的还是内核空间的。从而选择跳转到resume_kernel或resume_userspace。

4.9.1.2. Resuming a kernel control path

```
resume_kernel:
    cli                 ; these three instructions are
    cmpl $0, 0x14(%ebp) ; missing if kernel preemption
    jz need_resched     ; is not supported
restore_all:
    popl %ebx
    popl %ecx
    popl %edx
    popl %esi
    popl %edi
    popl %ebp
    popl %eax
    popl %ds
    popl %es
    addl $4, %esp
    iret
```

如果preempt_count为0，内核就跳转到need_resched处。否则，被中断的程序恢复运行。

4.9.1.3. Checking for kernel preemption

当这一步执行时，内核控制流肯定不是中断处理函数，否子，preempt_count不可能为0。如果被中断的是异常的话，那么内核控制流最多是两个异常流（其中一个已经结束了）。

如果TIF_NEED_RESCHED是0，就不需要进程切换。直接跳转到restore_all处。
如果需要进程切换，preempt_schedule_irq()函数被调用：设置PREEMPT_ACTIVE在preempt_count中，把大内核锁计数器置为-1，开中断，调用schedule()切换到另一个进程。如果之前的进程恢复了，preempt_schedule_irq()恢复大内核锁的计数器，清除PREEMPT_ACTIVE，关中断。

4.9.1.4. Resuming a User Mode program

```
resume_userspace:
  cli
  movl 0x8(%ebp), %ecx
  andl $0x0000ff6e, %ecx
  je restore_all
  jmp work_pending
```

4.9.1.5. Checking for rescheduling

```
work_pending:
   testb $(1<<TIF_NEED_RESCHED), %cl
   jz work_notifysig
work_resched:
   call schedule
   cli
   jmp resume_userspace
```

如果需要进程调度，schedule()执行，当该进程再次被调度回来，它将跳转回resume_userspace。

4.9.1.6. Handling pending signals, virtual-8086 mode, and single stepping

除了进程切换请求，还有如下工作需要进行。

```
work_notifysig:
    movl %esp, %eax
    testl $0x00020000, 0x30(%esp)
    je 1f
work_notifysig_v86:
    pushl %ecx
    call save_v86_state
    popl %ecx
    movl %eax, %esp
1:
    xorl %edx, %edx
    call do_notify_resume
    jmp restore_all
```

如果用户进程的eflags中VM置位，则调用save_v86_state()来在用户地址空间构建virtual-8086模式数据结构。接着调用do_notify_resume()函数来处理peding signal和single stepping。最后，跳转到restorw_all处，恢复被中断的进程。

Chapter 5. Kernel Synchronization

5.1. How the Kernel Services Requests

5.1.1. Kernel Preemption

抢占的主要特性是：运行在内核的进程在执行函数时，能被其它进程取代。（硬中断和软中断是会关抢占的）

当一个进程A运行异常处理函数时，另一个高优先级的进程B变成可运行。抢占式内核就可能在A的异常处理还未完成时就切换到B。相反的，在非抢占式内核，就不会有切换。例如，一个在执行异常处理的进程的时间片完了，如果是抢占式内核，进程马上会切换。如果是非抢占式内核，它就会接着执行下去。

可以抢占的条件是preempt_count为0，而3种情况之一发生会关抢占：
1． 中断处理函数中。

2. deferrable function（softirq或tasklet）执行中。
3. 内核显式的关闭抢占。

抢占只会发生在异常处理函数中，并且内核不能显式的关闭抢占。而且CPU必须开中断，否则强占不会执行（我觉的，只有开中断了，才会产生抢占的调用点）。

preempt_enable()也能产生抢占。当TIF_NEED_RESCHED置位，打开中断并且preempt_count为0,就会产生抢占。所以，内核抢占可能发生在：
1. 内核控制流结束时，比如中断处理完成。
2. 异常处理函数使用preempt_enable()重新开抢占时。
3. 开启deferrable function时。

5.1.2. When Synchronization Is Not Necessary

1. 所有的中断处理函数，向PIC确认中断，关闭了IRQ线。直到处理函数结束，都不会有相同的中断发生。
2. 中断处理，softirq,tasklet都是不可抢占，不可阻塞的。因此他们不可能被挂起很长时间。最坏情况下，它们只会被稍稍延时，因为其它中断的发生。
3. 中断处理函数不可能被deferrable function或系统调用中断。
4. Softirq和tasklet不可能在同一个CPU上嵌套。
5. 同样的tasklet不可能同时在不同CPU上执行。

一些简化设计的观点：
1. 中断处理和tasklet不需要被编码为可重入函数。
2. Per-CPU变量被softirq和tasklet访问不用同步。
3. 一种数据结构只被一种tasklet访问，不需要同步。

5.2. Synchronization Primitives

5.2.1. Per-CPU Variables

每个per-CPU变量只能被本CPU访问，因此就没有多CPU同步问题。但per-CPU变量倾向与内核抢占导致的竞争。一般的规则是，内核访问per-CPU变量必须关闭抢占，否则，如果内核控制流得到本CPU的变量地址，就被抢占了，然后迁移到另一个CPU上运行，变量地址却是前一个CPU的per-CPU变量的。

5.2.2. Atomic Operations

5.2.3. Optimization and Memory Barriers

当使用优化编译器，我们不能保证指令以在代码中的编写顺序执行。编译器会重新排序汇编语言，使之能更好的使用寄存器。而且，CPU会并行执行一些指令，可能会改变内存访问顺序。

在同步问题中，指令重排序必须避免。如果一条应该在同步原语之后执行的指令被优化后，排在了同步原语之前，那么事情将变的很麻烦。所以，所有的同步原语必须act as optimization and memory barriers。

一条optimization barrier原语使编译器不能通过寄存器传值来优化代码，而是每一步都得访问内存。但这条原语不能禁止编译器改变指令顺序。
一条memory barrier原语确保原语之前的操作执行完毕，才执行原语之后的操作。memory barrier像一个防火墙把操作隔离了，这样可以禁止指令重新排序。

5.2.4. Spin Locks

spinlock_t结构的最关键成员是:
slock:1代表解锁状态，0或负数代表加锁状态。
Break_lock:标志着一个进程正在忙等待这个锁。

5.2.4.1. The spin_lock macro with kernel preemption

spin_lock宏执行下列步骤:

1. 调用preempt_disable()关闭内核抢占。
2. 调用_raw_spin_trylock()，实现了对spinlock_t->slock的test-and-set原子操作。
3. 如果spin lock的旧值是正数，宏结束：内核控制流得到了spin lock。
4. 否则，内核没有得到spin lock，宏必须循环等待spin lock被释放。调用preempt_enable()取消第1步中增加的抢占计数器的值。如果在调用spin lock之前是开抢占的，那么现在就有一次抢占机会。
5. 如果break_lock等于0，把它置为1.通过检测该成员，spin lock的拥有者就能知道是否有其它进程在等待该锁。如果一个进程拥有spin lock太长，它就能决定提前放弃该锁，让其它正在等待的进程来执行。
6. 循环忙等待一小会。
7. 跳转到1，开始一次新的取锁过程。

5.2.4.2. The spin_lock macro without kernel preemption

略。

5.2.4.3. The spin_unlock macro

Spin_unlock宏简单的把slock置为1。然后调用preempt_enable()。

5.2.5. Read/Write Spin Locks

Read/Write spin lock用来增加内核的并发性。
read/write spin lock 的主要结构是rwlock_t，它的rwlock_t->lock是一个32-bit域，编码了两条信息:

1. 一个24-bit计数器代表了正在读取受保护数据结构的内核控制流数目。这个数目对2的补，存在0-24bit位上。
2. 一个标志位，如果没有控制流在读或写就置位，否则清零。这个标志位存放在第24bit位上。

例如: 0x01000000是spin lock idle。
0x00000000是指写者得到spin lock。
0x00ffffff，0x00fffffe分别是一个读者和两个读者。
rwlock_t->break_lock功能同spinlock_t。

5.2.5.1. Getting and releasing a lock for reading

read_lock宏的执行和spin_lock宏差不多。只是在第二步调用的是
_raw_read_lock():

```
int _raw_read_trylock(rwlock_t *lock)
{
   atomic_t *count = (atomic_t *)lock->lock;
   atomic_dec(count);
   if (atomic_read(count) >= 0)
       return 1;
   atomic_inc(count);
```

```
    return 0;
}
```

如果有一个写者，那么rwlock_t->lock==0x00000000,函数会返回0。

5.2.5.2. Getting and releasing a lock for writing

write_lock宏实现和spin_lock宏差不多。只是调用的是_raw_write_trylock():

```
int _raw_write_trylock(rwlock_t *lock)
{
    atomic_t *count = (atomic_t *)lock->lock;
    if (atomic_sub_and_test(0x01000000, count))
       return 1;
    atomic_add(0x01000000, count);
    return 0;
}
```

只有在rwlock_t->lock==0x01000000时才能得到锁。

5.2.6. Seqlocks

Seqlock给写者高优先权：写者能处理数据即使有读者正在读。好处是写者不用等待，坏处是读者有时得读取同一个值多次，直到取得了正确的拷贝。

seqlock_t结构：

1. struct spinlock_t lock。
2. sequence：每个读者必须读这个计数器2次，一次在读数据前，一次在读数据后。比较两个计数器值是否相等。一个写者在写数据前会递增该计数器。暗示读者数据必须重新读取。

write_seqlock()和write_sequnlock()都会递增seqlock_t->sequence的值。这确保了写者在写过程中，sequence是奇数，如果没有写者改变数据，它是偶数。

读者如下实现了临界区：

```
unsigned int seq;
do {
     seq = read_seqbegin(&seqlock);
     /* ... CRITICAL REGION ... */
} while (read_seqretry(&seqlock, seq));
```

注意到，当读者进入临界区时，不需要关闭抢占。换句话说，写者会关闭抢占，因为它使用了spin lock。

Seqlock只能保护下列数据：

1. 不能包含会被写者修改而被读者解引用的指针。因为写者可能使得指针失效，但读者如果正要访问该指针，将导致OOPs
2. 临界区里的读者不能有side effects。otherwise, multiple reads would have different effects from a single read)

5.2.7. Read-Copy Update (RCU)

Read-copy update (RCU)：对于被RCU保护的共享数据结构，读者不需要获得任何锁就可以访问它，但写者在访问它时首先拷贝一个副本，然后对副本进行修改，最后使用一个回调（callback）机制在适当的时机把指向原来数据的指针重新指向新的被修改的数据。这个时机就是所有引用该数据的CPU都退出对共享数据的操作。

RCU的局限性：

1. 只有动态分配的数据结构并被指针引用，才能被RCU保护。
2. 被RCU保护的控制流不能睡眠。

当一个内核控制流要读取被RCU保护的数据结构前，调用rcu_read_lock(),它等同于preempt_disable()，接着读者解引用指针开始读取数据结构。读者不能够睡眠，直到数据读取完成。最后rcu_read_unlock()等价于preempt_enable();

旧的数据在所有读者都调用rcu_read_unlock()后被释放，内核要求读者在下列情况发生前调用rcu_read_unlock():

1. CPU进程切换。
2. CPU开始在在用户模式执行。
3. CPU执行idle loop。

In each of these cases, we say that the CPU has gone through a quiescent state.

call_rcu()函数被写者用来处理旧的数据。

5.2.8. Semaphores

semaphores会使进程睡眠，因此中断处理和deferrable function不能使用它。

semaphores数据结构是,struct semaphore:

1. count:如果大于0，说明资源可用。等于0说明资源用完但没有别的进程在等待。最后，如果小于0，资源不可用并且至少有一个进程在等待它。
2. wait: 等待该资源的等待队列，如果count大于等于0，等待队列为空。
3. sleepers: 标志，指出是否有一些进程在semaphore上睡眠。

down操作的3种典型的情况(记住sleeper只能为0或1):

1. MUTEX semaphore open (count equal to 1, sleepers equal to 0):
   count置0，运行临界区代码。
2. MUTEX semaphore closed, no sleeping processes (count equal to 0, sleepers equal to 0)
   down宏递减count为-1，调用__down()，此时count为-1，sleeper为0。每次迭代，__down()检查count是否为负:
   如果count为负，调用schedule()挂起进程，count为-1，sleeper为1。当被唤醒时，接着迭代。
   如果count非负，设置sleeper为0，退出迭代。它尝试唤醒另一个在等待队列中的另一个进程（在我们这个场景中，等待队列为空）,带着资源结束运行。退出时，count和sleepers都为0。
3. MUTEX semaphore closed, other sleeping processes (count equal to -1, sleepers equal to 1)

   The down macro decreases count and invokes the _ _down( ) function with count set to -2 and sleepers set to 1. The function temporarily sets sleepers to 2, and then undoes the decrement performed by the down macro by adding the value sleepers-1 to count. At the same time, the function checks whether count is still negative (the semaphore could have been released by the holding process right before _ _down( ) entered the critical region).

   If the count field is negative, the function resets sleepers to 1 and invokes schedule( ) to suspend the current process. The count field is still set to -1, and the sleepers field to 1.

If the count field is not negative, the function sets sleepers to 0, tries to wake up another process in the semaphore wait queue, and exits holding the semaphore. On exit, the count field is set to 0 and the sleepers field to 0. The values of both fields look wrong, because there are other sleeping processes. However, consider that another process in the wait queue has been woken up. This process does another iteration of the loop; the atomic_add_negative( ) function subtracts 1 from count, restoring it to -1; moreover, before returning to sleep, the woken-up process resets sleepers to 1.

5.2.8.1. Getting and releasing semaphores

up()用来释放资源。up()增加count,如果count大于0，说明没有睡眠进程，否则，唤醒睡眠着的进程。

down()用来获取资源。down()减少count，如果大于等于0，当前进程得到资源。否则,count是负数，当前进程必须被挂起，一些寄存器的值被保存在栈中，调用__down()本质上，_down()把当前进程状态由TASK_RUNNING变到TASK_UNINTERRUPTIBLE，挂入等待队列。使用spin_lock_irqsave（会关闭中断,保护等待队列）操作， sem->wait.lock spin lock保护semaphore wait queue。

5.2.9. Read/Write Semaphores

略。

5.2.10. Completions

completion()和wait_for_completion()配对使用。

completion()和wait_for_completion()确保不会同时执行，但up()和down()可能会同时执行。

5.2.11. Local Interrupt Disabling

关中断用来保护被中断处理函数使用的数据，但它不能阻止在其它CPU上运行的中断访问共享数据，因此它常与spin lock同时使用。

5.2.12. Disabling and Enabling Deferrable Functions

deferrable function能在不可预料的时间执行（本质上，是在中断处理函数完成后）。那么，被deferrable function访问的数据结构要被保护。

利用preempt_count实现deferrable function的开与关。do_softirq和tasklet当preempt_count为正时不执行。

local_bh_disable()递增preempt_count，local_bh_enabl （）递减preempt_count，因此local_bh_disable()可以被嵌套很多次。

local_bh_disable()两个重要工作,帮助等待较长的线程尽快执行:

1. 检测preempt_count的硬中断计数器和软中断计数器，如果为0，并且有pending softirq，则调用do_softirq()。
2. 如果有TIF_NEED_RESCHED被标记，则调用preempt_schedule()看是否能抢占。

5.4. Examples of Race Condition Prevention

5.4.1. Reference Counters

5.4.2. The Big Kernel Lock

大内核锁本质上也是自旋锁，但是它又不同于自旋锁，自旋锁是不可以递归获得锁的，因为那样会导致死锁。但大内核锁可以递归获得锁。大内核锁用于保护整个内核，而自旋锁用于保护非常特定的某一共享资源。进程保持大内核锁时可以发生调度，具体实现是：在执行schedule时，schedule将检查进程是否拥有大内核锁，如果有，它将被释放，以致于其它的进程能够获得该锁，而当轮到该进程运行时，再让它重新获得大内核锁。注意在保持自旋锁期间是不运行发生调度的。
需要特别指出，整个内核只有一个大内核锁，其实不难理解，内核只有一个，而大内核锁是保护整个内核的，当然有且只有一个就足够了。
还需要特别指出的是，大内核锁是历史遗留，内核中用的非常少，一般保持该锁的时间较长，因此不提倡使用它。从2.6.11内核起，大内核锁可以通过配置内核使其变得可抢占（自旋锁是不可抢占的），这时它实质上是一个互斥锁，使用信号量实现。

Chapter 6. Timing Measurements

6.1. Clock and Timer Circuits

6.1.1. Real Time Clock (RTC)

RTC独立于CPU和其他所有芯片。它整合在SMOS RAM中。Linux仅仅使用RTC获取时间和日期。

Linux可以利用RTC来得到比Programmable Interval Timer更加精确的时间测量方案。

6.1.2. Time Stamp Counter (TSC)

所有的80X86微处理器可以接收外部时钟信号。64bit的的TSC寄存器，每当一个时钟信号，会递增1。

6.1.3. Programmable Interval Timer (PIT)

PIT不停的以固定的频率（内核指定）发送时钟中断。Linux指定为1000-Hz频率，大约是每1个milliseconde，也被叫做一个tick。

6.1.4. CPU Local Timer

Local APIC类似于PIT,又少量不同：
1.APIC的timer counter是32 bit长，而PIT的是16bit。因此它能发送很低频率的时钟中断(counter保存着两次时种中断间隔的ticks)。
2.APIC的定时器只发送中断给local CPU,而PIT是全局的。
3.APIC定时器是基于bus clock 信号。

6.1.5. High Precision Event Timer (HPET)

略。

6.2. The Linux Timekeeping Architecture

6.2.1. Data Structures of the Timekeeping Architecture

6.2.1.1. The timer object

代表了定时器的源。timer_opts数据结构，关键成员：
1． name:定时器源名字。

2. make_offset:记录最近一次tick的时间，被时钟中断处理函数使用。
3. get_offset: 返回从最近一次tick以来经过的时间。

使用make_offset和get_offset方法，Linux内核得到了更为精准的时间。这个操作叫time interpolation。

cur_timer变量存放了最佳可用定时器源的地址。

6.2.1.2. The jiffies variable

jiffies变量保存了从系统开始以来经过的tick数。在时钟中断发生时递增。它是jiffies_64变量的低32bit。
在32bitCPU中，longlong不是原子操作，为了增加效率，所以把jiffies设为32bit。

6.2.1.3. The xtime variable

Xtime变量保存了当前时间和日期。主要是timespec数据结构:

tv_sec: Stores the number of seconds that have elapsed since midnight of January 1, 1970 (UTC)。

tv_nsec: 存放Stores the number of nanoseconds that have elapsed within the last second (its value ranges between 0 and 999,999,999)

6.2.2. Timekeeping Architecture in Uniprocessor Systems

在单CPU系统中，所有与时间相关的活动在PIC在IRQ线0触发时钟中断。Linux中，在时钟中断发生时会处理一些动作，另一些则被推迟到deferrable function中执行。

6.2.2.1. Initialization phase

在系统初始化阶段，time_init()函数被调用，来建立timekeep结构。它主要是调用select_timer()选择最佳的定时器源，并把相应对象地址存放在cur_timer变量中。调用setup_irq(0,&irq0)建立中断处理。

6.2.2.2. The timer interrupt handler

timer_interrupt()函数是PIT或HPET的ISR。执行步骤:

1. 保护与时间相关的内核变量，write_seqlock () 在xtime_lock seqlock。
2. 执行cur_timer的mark_offset方法，主要动作是检测从上一次tick开始，是否有中断丢失，如果有，则更新jiffies_64值。
3. 调用do_timer_interrupt()函数，它执行下列动作:
   a.把jiffies_64加1.
   b.调用update_times()更新系统日期和时间，计算系统负载。
   c.调用update_process_times()函数，执行一些与时间相关的统计。
   d.调用profile_tick()。
   e.如果系统时钟被外部时钟同步，调用set_rtc_mmss()更新RTC。这个特性帮助系统在网络上更新时钟。
4. 利用write_sequnlock()放弃xtime_lock seqlock。
5. 返回值1

6.2.3. Timekeeping Architecture in Multiprocessor Systems

SMP系统有两种时钟中断源：被PIT或HPET产生的，被CPU local timers。

6.2.3.1. Initialization phase

所有的APICd定时器被同一个系统总线同步。这意味着calibrate_APIC_clock()计算出的值，对所有CPU有效。

6.2.3.2. The global timer interrupt handler

SMP版本的timer_interrupt()函数与UP版本有少许不同：

1. do_timer_interrupt()，把中断确认写入I/O APIC。
2. update_process_times()函数不再使用，因为它的动作与特定CPU相关。
3. profile_tick()不调用，因为它的动作与特定CPU相关。

6.2.3.3. The local timer interrupt handler

这个处理函数用来timekeep与特定CPU相关的活动，检测当前进程在指定CPU上运行时间。

6.3. Updating the Time and Date

```
void update_times(void)
{
   unsigned long ticks;
   ticks = jiffies - wall_jiffies;
   if (ticks) {
       wall_jiffies += ticks;
       update_wall_time(ticks);
   }
   calc_load(ticks);
}
```

wall_jiffies存储着上一次跟新xtime的时间。wall_jiffies的值可以小于jiffies-1，因为一些时钟中断可能会丢失。而且，内核也不需要每次tick都更新xtime。然而，没有tick最终会丢失，在长时间运行后，xtime存储了正确的系统时间。

6.5. Software Timers and Delay Functions

Linux有两种定时器：dynamic timers（用于内核），interval timers（用于用户空间）。
定时器不适合用在实时应用中。

6.5.1. Dynamic Timers

dynamic timers动态的创建和销毁，没有数量限制。
主要数据结构是，timer_list：

```
struct timer_list {
  struct list_head entry;
  unsigned long expires;
  spinlock_t lock;
  unsigned long magic;
  void (*function)(unsigned long);
  unsigned long data;
  tvec_base_t *base;
};
```

在Linux2.6，dynamic timer绑定在activate它的CPU中，定时器函数总是在执行了add_timer( )或the mod_timer( )的CPU上。但del_timer( )能够deactivate任何一个dynamic timer。

6.5.1.1. Dynamic timers and race conditions

6.5.1.2. Data structures for dynamic timers

管理dynamic timer的主要数据结构是per-CPU变量tvec_bases[NR_CPUS]。

元素类型tvec_base_t，挂载了所有要处理的绑定在该CPU上的定时器。

```
typedef struct tvec_t_base_s {
    spinlock_t lock;
    unsigned long timer_jiffies;
    struct timer_list *running_timer;
    tvec_root_t tv1;
    tvec_t tv2;
    tvec_t tv3;
    tvec_t tv4;
    tvec_t tv5;
} tvec_base_t;
```

tv1 ：包含一个有256个list_head的数组。链接着在在下255个ticks将会超时的定时器。

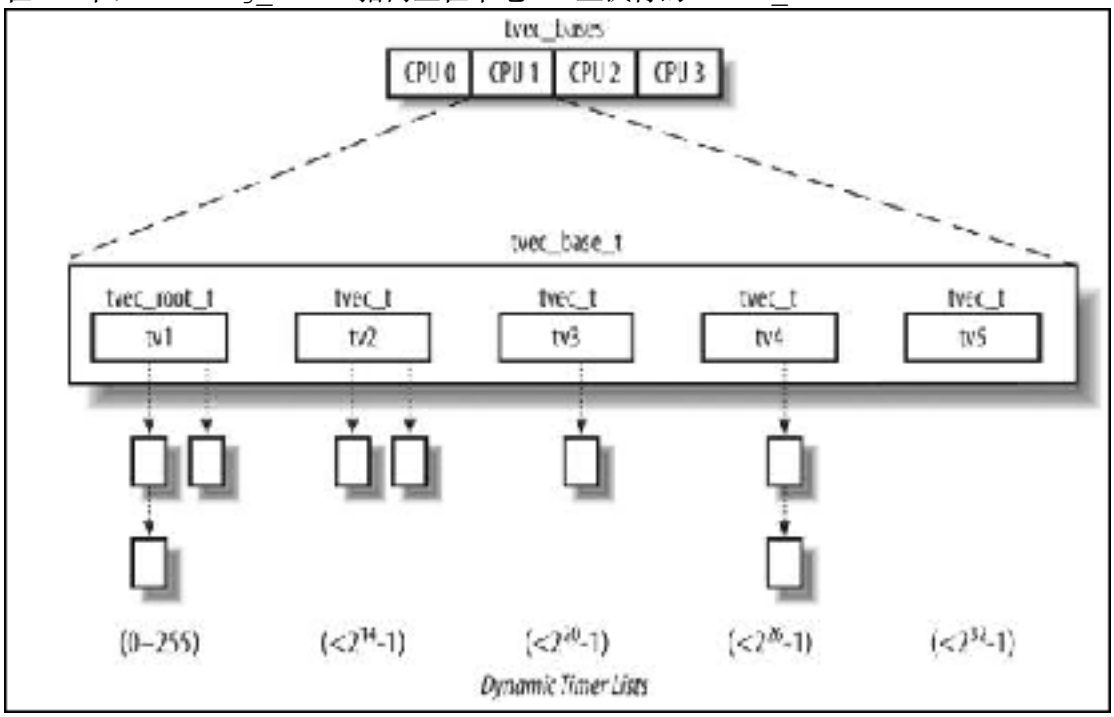
tv2,tv3,tv4 ：包含有一个64个list_head的数组。链接着将在2^{14}-1, 2^{20}-1, 和 2^{26}-1个ticks超时的定时器。

tv5：与之前的类似，只是tv5的中的数组的最后一项链接着所有超时时间极端大的定时器。而且tv5不需要从其它数组补充定时器。

timer_jiffies存放的是上一次定时器超时的时间：如果它等于jiffies，no backlog of deferrable functions has accumulated，如果小于jiffies，then lists of dynamic timers that refer to previous ticks must be dealt with.

timer_jiffies仅仅被run_timer_softirq()函数递增。如果定时器超时处理函数长时间未被执行（中断过多，或deferrable function被关闭了），那么timer_jiffies会比jiffies小很多。

在SMP中，running_timer指向正在本地CPU上执行的timer_list。



6.5.1.3. Dynamic timer handling

定时器处理是比较耗时的，所以不能放在中断处理中。2.6内核中，定时器处理放在TIMER_SOFTIRQ softirq中。

run_timer_softirq()是TIMER_SOFTIRQ softirq的响应函数。执行步骤:

1. 保存本地CPU的tvec_base_t数据结构地址到base局部变量中。
2. 获取base->lock spin lock和关中断。
3. 循环，直到base->timer_jiffies大于jiffies。每次循环中执行下列动作:
   f.计算base->tv1中将要被处理的下一个定时器链表的index.
   ```
   index = base->timer_jiffies & 255;
   ```
   g.如果index为0，说明所有的base->tv1都被检查过了，函数调用cascade()过滤定时器。
   ```
   if (!index &&(!cascade(base, &base->tv2, (base->timer_jiffies>>
   8)&63)) &&
   (!cascade(base, &base->tv3, (base->timer_jiffies>>14)&63)) &&
   (!cascade(base, &base->tv4, (base->timer_jiffies>>20)&63)))
         cascade(base, &base->tv5, (base->timer_jiffies>>26)&63);
   ```
   cascade()函数：接收bsae中地址，base->tv2的地址，base->tv2中将在下256个ticks超时的定时器链表的index。cascade()将tv2中合适的定时器移动到tv1中。
   h.把base->timer_jiffies递增1。
   i.对每一个在base->tv1.vec[index]中的定时器，执行相应的定时器函数。对每一个timer_list元素t执行下列步骤:
      1. 从base->tv1中移除t.
      2. 在SMP中，把base->running_timer设为t.
      3. t.base设为NULL.
      4. 释放base->lock，开终端。
      5. 执行t.function，参数为t.data.
      6. 得到base->lock，关中断。
      7. 继续下一个定时器。
4. 所有链表中的定时器被执行完毕，接着3的循环。
5. 循环结束，所有定时器处理完成。在SMP中，把base->running_timer设为NULL。
6. 释放base->lock spin lock，开中断。

定时器算法很复杂，但有很好的性能。To see why, assume for the sake of simplicity that the TIMER_SOFTIRQ softirq is executed right after the corresponding timer interrupt occurs. Then, in 255 timer interrupt occurrences out of 256 (in 99.6% of the cases), the run_timer_softirq( ) function just runs the functions of the decayed timers, if any. To replenish base->tv1.vec periodically, it is sufficient 63 times out of 64 to partition one list of base->tv2 into the 256 lists of base->tv1. The base->tv2.vec array, in turn, must be replenished in 0.006 percent of the cases (that is, once every 16.4 seconds). Similarly, base->tv3.vec is replenished every 17 minutes and 28 seconds, and base->tv4.vec is replenished every 18 hours and 38 minutes. base->tv5.vec doesn't need to be replenished.

Chapter 7. Process Scheduling

7.1. Scheduling Policy

在Liunx中,进程优先级是动态变化的，等待很久的进程会提升优先级，运行很长时间的进程会降低优先级。一般分为3种进程：

1. Interactive processes:与用户交互的进程，需要快的响应时间。
2. Batch processes: 一般在后台运行，不需要很快的响应时间。
3. Real-time processes: 不会被低优先级进程阻塞，有确定的响应时间。

7.1.1. Process Preemption

Linux进程是可抢占的，当一个进程进入TASK_RUNNING状态，内核检测它的动态优先级是否大于当前进程的优先级，如果是，当前进程就被高优先级进程抢占了。一个进程也能在时间片用完时被抢占，在时间中断处理函数中把时间片用完的进程置TIF_NEED_REDCHED，当中断结束后就能发生调度。

7.2. The Scheduling Algorithm

2.6的调度器的代价仅O(1)。调度器总能找到一个进程来执行，PID 0进程永远存在，SMP中每一个CPU都有自己的PID 0进程。

3种调度分类：

1. SCHED_FIFO: A First-In, First-Out real-time process,当没有更高优先级的进程时，该进程将运行希望运行的时间。即使有同优先级的进程存在，也不能抢占它。
2. SCHED_RR: A Round Robin real-time process,同优先级的进程之间可以轮流执行。
3. SCHED_NORMAL:常规的，分时进程。

7.2.1. Scheduling of Conventional Processes

每一个常规进程有自己的static priority，范围从100(最高优先级)到139(最低优先级)。

一个新进程总是继承它父进程的static priority，static priority可以用nice()和setpriority()系统调用改变。

7.2.1.1. Base time quantum

static priority本质上决定了一个进程的base time quantum，它指的是在一个进程时间片用完后，新赋给它的时间片。static priority和base time quantum的关系如下面公式：

$$\text{base time quantum (in milliseconds)} = \begin{cases} (140 - \text{static priority}) \times 20 & \text{if static priority} < 120 \\ (140 - \text{static priority}) \times 5 & \text{if static priority} \geq 120 \end{cases} \quad (1)$$

可见，越高的static priority，越常的base time quantum。结果是，高优先级进程比低优先级进程得到更长的时间片。

7.2.1.2. Dynamic priority and average sleep time

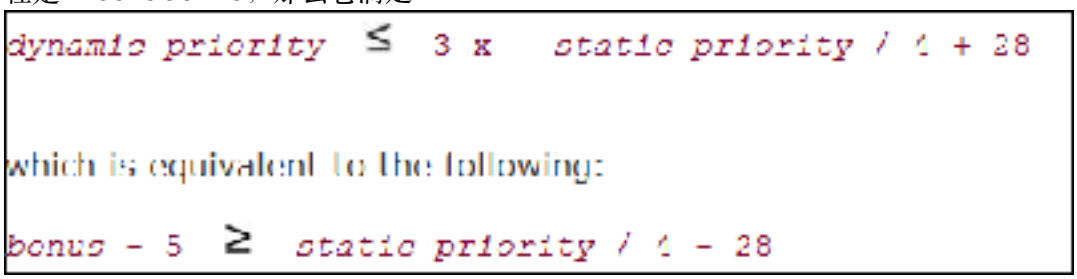
进程还有一个dynamic priority，范围从100(最高)到139(最低)。dynamic priority是调度器真正用来选择进程运行的数值。它与static priority的关系满足：

$$\text{dynamic priority} = \max(100, \min(\text{static priority} - \text{bonus} + 5, 139)) \quad (2)$$

Bouns的范围是0到10；比5小的值代表对低dynamic priority进程的惩罚，大于5的数值是对dynamic priority的奖励。bouns的值，依赖与进程的历史，更精确的说，它依赖于进程的平均睡眠时间。

粗略的说，平均睡眠时间是进程睡眠所花费的平均nanosecond数目。但要注意，它不是子过去时间里的平均操作时间。比如，在TASK_INTERRUPTIBLE状态中睡眠和在TASK_UNINTERRUPTIBLE状态中睡眠，对平均睡眠时间的计算影响是不同的。而且，当进程运行时，平均睡眠时间会减少，平均睡眠时间也不会大于1秒。

平均睡眠时间也被调度器用来判断一个进程是否是interactive或batch。如果一个进程是interactive，那么它满足：



可见，高优先级进程更容易被看成是interactive进程。优先级139的进程则完全不可能成为interactive进程。

7.2.1.3. Active and expired processes

高static priority的常规进程不能完全锁住低static priority进程。一个进程时间片结束后，它会被一个时间片还未用完的低优先级进程取代。调度器维持了2个运行进程集合：
Active processes:没有消耗完时间片的可运行进程。
Expired processes：消耗完时间片的进程，它们被禁止运行，直到所有active processes结束时间片。

实际的算法要复杂些，调度器希望增强interactive进程的性能。一个active batch进程在时间片耗完后就变成expired，一个interactive进程消耗完时间片后仍然是active：调度器填充它的时间片，把它留在active集合中。然而，当最老的expired进程等待了过长时间时，或一个expired进程有比interactive进程更高的优先级，interactive也会被移动到expired集合。这样，active集合会变空，expired集合里的进程也有机会运行。

7.2.2. Scheduling of Real-Time Processes

每一个实时进程和一个real-time priority联系，范围从1(最高)到(99)最低。调度器总是让高优先级进程运行。高优先级进程抑制了低优先级进程的运行。

一个实时进程被取代仅发生在如下情况：
1. 被告优先级进程抢占。
2. 进程执行了阻塞操作，进入睡眠。
3. 被stopped或killed。
4. 通过系统调用主动放弃CPU。
5. SCHED_RR策略下，时间片消耗完。

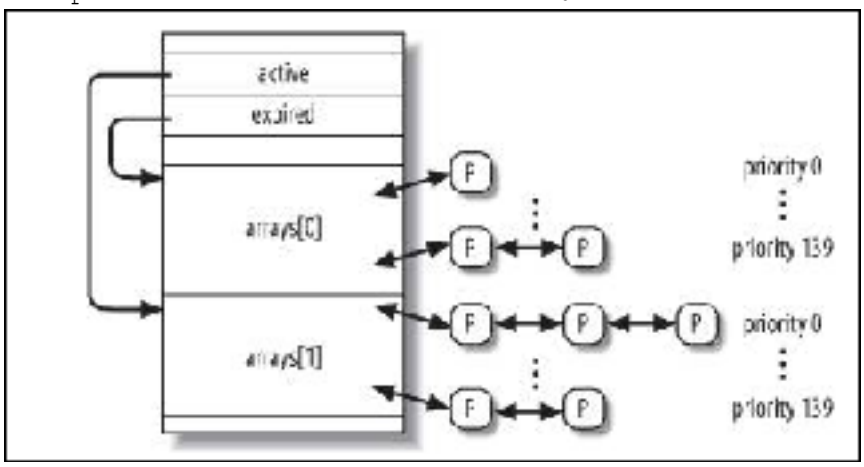
7.3. Data Structures Used by the Scheduler

runqueue链接了所有TASK_RUNNING进程，除了0号进程。

7.3.1. The runqueue Data Structure

per-CPU变量runqueues，数据结构是runqueue，每个CPU都有自己的runqueue结构。

每一个可运行进程都属于唯一一个runqueue, runqueue->arrays由两个prio_array_t结构组成。每个prio_array_t结构代表一个可运行队列集合。每个prio_array_t结构有:140个list head, a priority bitmap, a counter of the processes included in the set。



7.3.2. Process Descriptor

当一个新进程被创建，sched_fork( ), invoked by copy_process( ),设置current->time_slice和p(孩子)进程的时间片:

```
p->time_slice = (current->time_slice + 1) >> 1;
current->time_slice >>= 1;
```

如果一个进程在它第一个时间片还没结束就终止了，会把剩下的时间片奖励给父进程。

7.4. Functions Used by the Scheduler

The scheduler relies on several functions in order to do its work; the most important are:

scheduler_tick( )
Keeps the time_slice counter of current up-to-date

try_to_wake_up( )
Awakens a sleeping process

recalc_task_prio( )
Updates the dynamic priority of a process

schedule( )
Selects a new process to be executed

load_balance()
keeps the runqueues of a multiprocessor system balanced

7.4.1. The scheduler_tick( ) Function

它的执行步骤：

1. 储存TSC值到runqueue->timestamp_last_tick。
2. 检查当前进程是否是0号进程。如果是，则：
   a. 如果local runqueue包含了其它可运行进程，设置TIF_NEED_RESCHED，强制进行调度。
   b. 跳到7（不需要更新0号进程的时间片）。
3. 检查curent->array指针是否指向active list。如果不是，进程消耗完了时间片但还未被替换，设置TIF_NEED_RESCHED强制调度。跳到7.
4. 得到this_rq()->lock spin lock。
5. 减少当前进程的时间片，检查时间片是否消耗完了。
6. 放弃this_rq()->lock spin lock。
7. 调用rebalance_tick()函数，确保每个CPU的runque大小基本相同。

7.4.1.1. Updating the time slice of a real-time process

如果进程是FIFO实时进程，scheduler_tick()不做啥。当前进程也不能被优先级小于等于它的进程抢占，因此没必要更新时间片。

如果是RR实时进程，scheduler_tick()递减时间片，并检查时间片是否用完了。如果用完了，函数将让当前进程尽量被抢占。

7.4.1.2. Updating the time slice of a conventional process

如果是常规进程，scheduler_tick()函数执行下列步骤：

1. 递减时间片(current->time_slice)。
2. 检测时间片，如果消耗完了，执行下列操作：
   a. 调用dequeue_task()从active集合中，移走当前进程。
   b. 调用set_tsk_need_resched()设置TIF_NEED_RESCHED。
   c. 更新当前进程的dynamic priority。
   d. 重新填充时间片。
   e. 如果runqueue->expired_timestamp等于0（expired processes集合为空），把当前tick写入。
   f. 把当前进程插入active集合或expired集合：

```
      if(!TASK_INTERACTIVE(current)||
EXPIRED_STARVING(this_rq()) {
enqueue_task(current, this_rq( )->expired);
 if (current->static_prio < this_rq( )->best_expired_prio)
    this_rq( )->best_expired_prio = current->static_prio;
} else
   enqueue_task(current, this_rq( )->active);
```

   TASK_INTERACTIVE：如果是interactive进程则返回1.
   EXPIRED_STARVING：如果最先消耗完时间片的进程已等待时间>=1000*runqueue进程数目+1，那么返回1，或者当前进程的静态优先级比runqueue中expired进程优先级低。
3. 否则，如果时间篇没有耗完，检查当前进程是否剩下的时间片太长了。

7.4.2. The try_to_wake_up( ) Function

这个函数唤醒睡眠或暂停的进程，把它置为TASK_RUNNING，插入local CPU的runqueue。例如，函数能唤醒在等待队列中的进程或恢复正在等待信号的进程运行。
参数：要唤醒的进程描述符（p）；
能唤醒的进程状态(state)；

标志位（sync），禁止唤醒的进程抢占正在运行的进程。

它的运行步骤是：

1. 调用task_rq_lock()关中断，acquire the lock of the runqueue rq owned by the CPU that was last executing the process (it could be different from the local CPU).
2. 检测p->state是否属于参数state。如果不属于，跳转到9，中止函数。
3. 如果p->array非NULL，说明进程已经属于runqueue，那么跳转到8。
4. 在SMP，检测被唤醒的进程是否要从最后一次执行所在CPU迁移到另一个CPU。函数根据启发式规则选择目标runqueue。
5. 如果进程是在TASK_UNINTERRUPTIBLE，那么递减目标runqueue->nr_uninterruptible值，设置p->activated值为-1（表示被唤醒进程是从TASK_UNINTERRUPTIBLE进入TASK_RUNNING）。
6. 调用activate_task()：
   a. 调用sched_clock()得到当前timestamp。如果目标CPU不是当前CPU,要做些调整。
   ```
   now = (sched_clock( ) - this_rq( )->timestamp_last_tick)
                 + rq->timestamp_last_tick;
   ```
   b. 调用recalc_task_prio();
   c. 设置p->activated.
   d. 设置p->timestamp为now。
   e. 把进程描述符插入active集合。
7. 如果要么目标CPU不是local CPU或者sync不为1，检测新可运行进程是否有高优先级，那么就可以抢占了。如果是SMP，就要使用IPI来使目标CPU抢占。
8. p->state置为TASK_RUNNING。
9. 调用task_rq_unlock()解锁，开中断。
10. 成功唤醒返回1，否则为0。

7.4.3. The recalc_task_prio( ) Function

这个函数更新平均睡眠时间，和一个进程的dynamic priority。
略。

7.4.4. The schedule( ) Function

schedule()函数实现了调度器。It is invoked, directly or in a lazy (deferred) way, by several kernel routines

7.4.4.1. Direct invocation

当前进程在需要的资源不可用时必须阻塞，因此会直接调用调度器。一般步骤是：

1. 把current插入合适的wait queue。
2. 把current的状态转换成TASK_INTERRUPTIBLE或TASK_UNINTERRUPTIBLE。
3. 调用schedule();
4. 检测资源是否可用了；如果不可用，跳转到2.
5. 如果可用了，把current从wait queueu移除。

7.4.4.2. Lazy invocation

通过设置current的TIF_NEED_RESCHED位，进行lazy 调用。在返回用户空间前，会检测该位，从而调用schedule()。

典型的lazy调用有：

1．当current用完了CPU时间片。
2．进程被唤醒，并且优先级比当前进程高。
3．系统调用强制调度。

7.4.4.3. Actions performed by schedule( ) before a process switch

schedule()关闭了内核抢占。
schedule()维持了大内核锁：

```
if (prev->lock_depth >= 0)
    up(&kernel_sem);
```

它不改变lock_depth,只是释放了资源，当切换回来时，发现lock_depth非负，就再次得到大内核锁。

schedule()检测prev状态，如果它不是可运行的，**而且还没有被抢占**，那么就把它从runqueue中移除。如果有信号pending并且TASK_INTERRUPTIBLE，那么把它置为TASK_RUNNING并插入runqueue。这并不是把CPU交给prev，只是给它一个运行的机会。

```
if (prev->state != TASK_RUNNING &&
    !(preempt_count() & PREEMPT_ACTIVE)) {
    if (prev->state == TASK_INTERRUPTIBLE &&
signal_pending(prev))
        prev->state = TASK_RUNNING;
    else {
        if (prev->state == TASK_UNINTERRUPTIBLE)
            rq->nr_uninterruptible++;
        deactivate_task(prev, rq);
    }
}
```

schedule检查runqueue中的可运行进程数目。如果有可运行进程，函数就调用dependent_sleeper()函数，它一般返回0，除非CPU支持超线程。

如果没有可运行进程存在，函数调用idle_balance()从别的runqueue转移一些进程来local runqueue。

如果idle_balance()失败，则把0号进程调度进来。

如果active集合没有进程了，就把active和expired指针对换。

在prio_array_t数据结构中查找一个可运行进程。首先，schedule()在位图中查找第一个非0元素，然后在对应的优先级队列中取得第一个进程描述符。

next局部变量保存了将要运行的进程。schedule()查看next->activated，这个预编码了被唤醒进程的状态：

- 0: The process was in TASK_RUNNING state.
- 1: The process was in TASK_INTERRUPTIBLE or TASK_STOPPED state, and it is being awakened by a system call service routine or a kernel thread.
- 2: The process was in TASK_INTERRUPTIBLE or TASK_STOPPED state, and it is being awakened by an interrupt handler or a deferrable function.

- -1: The process was in TASK_UNINTERRUPTIBLE state and it is being awakened.

如果进程是从TASK_INTERRUPTIBLE or TASK_STOPPED状态被唤醒，调度器增加进程的平均睡眠时间。

```
if (next->prio >= 100 && next->activated > 0) {
    unsigned long long delta = now - next->timestamp;
    if (next->activated == 1)
        delta = (delta * 38) / 128;
    array = next->array;
    dequeue_task(next, array);
    recalc_task_prio(next, next->timestamp + delta);
    enqueue_task(next, array);
}
next->activated = 0;
```

可见，调度器对被中断或defferable function唤醒的进程和被系统调用或内核线程唤醒的进程区别对待。对前者，调度器把整个runqueue等待时间加上，对后者，只加了一部分。这是因为interactive进程更可能被异步事件唤醒些。

7.4.4.4. Actions performed by schedule( ) to make the process switch

schedule()已经决定好了要运行的下一个进程。内核将访问next的thread_info数据结构。它的地址在next的进程描述符:

```
switch_tasks:
prefetch(next);
```

prefetch宏暗示CPU控制单元把next的第一个成员的内容放在硬件cache中，它是并行执行的。

清除TIF_NEED_RESCHED位，如果schedule()是lazy调用。

context_switch()函数建立了next的地址空间。它确保了即使next是内核线程，它将使用prev的地址空间。如果next是常规进程，则使用自己的地址空间。最后调用swich_to()完成切换。

7.4.4.5. Actions performed by schedule( ) after a process switch

The instructions of the context_switch( ) and schedule( ) functions following the switch_to macro invocation will not be performed right away by the next process, but at a later time by prev when the scheduler selects it again for execution. However, at that moment, the prev local variable does not point to our original process that was to be replaced when we started the description of schedule( ), but rather to the process that was replaced by our original prev when it was scheduled again. (If you are confused, go back and read the section "Performing the Process Switch" in Chapter 3.)

The very last instructions of the schedule( ) function are:

```
finish_schedule:
prev = current;
if (prev->lock_depth >= 0)
    __reacquire_kernel_lock( );
preempt_enable_no_resched();
if (test_bit(TIF_NEED_RESCHED, &current_thread_info( )->flags)
   goto need_resched;
return;
```

可见,schedule()如果有必要，将重新得到大内核锁，重开抢占，检查是否有其它进程把当前进程设置了TIF_NEED_SECHED，如果设置了，schedule()将从头开始执行。

7.5. Runqueue Balancing in Multiprocessor Systems

7.5.1. Scheduling Domains

本质上，一个scheduling domain是一个CPU集合，它们的负载可以被内核保持平衡。一般来说，scheduling domain是分层组织的：最顶层的scheduling domain通常包括所有CPU，包含了孩子scheduling domain,孩子scheduling domain包含了CPUs的子集。

每个scheduling domain被划分1个或多个groups，每个group代表了一个scheduling domain的CPU子集。负载平衡在一个scheduling domain的group中进行。换句话说，进程从一个CPU移动到另一个，当且仅当同一个scheduling domain中一些group的负载低于另一个group。

每个scheduling domain被一个sched_domain descriptor表示，每个scheduling domain的一个组被一个sched_group descriptor表示。sched_group->groups指向一个组描述符链表的第一个元素。sched_group->parent指向父scheduling domain描述符。

所有CPU的sched_domain descriptor存储在一个per-CPU变量phys_domains中。如果内核不支持超线程技术，这些domains都在domains hierarchy的底部。runqueue->sd指向它们，它们是base scheduling domain。

7.5.2. The rebalance_tick( ) Function

rebalance_tick()被scheduler_tick()调用，每次tick一次。它确定runqueue中的进程数目，更新runqueue的平均负载。它访问runqueue->nr_running和runqueue->cpu_load。

rebalance_tick()循环遍历scheduling domain，从base domain(runqueue->sd指向)到顶层domain。每次迭代，函数决定是否调用load_balance()函数，它会在scheduling domain上执行平衡操作。

7.5.3. The load_balance( ) Function

load_balance()函数检测一个scheduling domain是否显著的不平衡，进一步说，它检测是否能通过移动一些进程来减少不平衡。

1. 得到this_rq->lock spin lock。
2. 调用find_busiest_group()分析scheduling domain里的groups的负载。函数返回最忙碌的group的sched_group描述符地址，假如它不包含当前CPU。在这种情况下，函数也返回需要移动到本CPU上的进程个数。相反地，如果最忙的组包含了local CPU或所有组基本平衡，它就返回NULL。This procedure is not trivial, because the function tries to filter the statistical fluctuations in the workloads.
3. 如果find_busiest_group()找不到合适的组，它将释放spin lock，调整scheduling domain descriptor的参数，使之推迟下一次find_busiest_group()在local CPU的调用，然后结束load_balance()函数。
4. 调用find_busiest_queue()函数找到最忙的CPU（在2中找到的组中寻找）。它返回最忙的CPU的runqueue->busiest。
5. 请求第二个spin lock，叫做busiest->lock spin lock。

6. 调用move_task()函数，从busiest runqueue中移动一些进程到local runqueue。
7. 如果move_task()失败，scheduling domain仍然是不平衡的。把busiest->active_balance置为1，唤醒migration内核线程。这个内核线程遍历scheduling domain链，寻找一个idle CPU。如果找到了，内核线程调用move_task()移动一个进程到idle runqueue。
8. 释放spin lock。
9. 结束函数。

7.5.4. The move_tasks( ) Function

move_tasks()函数移动进程。它先分析源runqueue，从expired集合遍历，再是active集合。对每一个进程调用can_migrate_task()，这个函数返回1的条件：

1. 进程不能是正在被remote CPU执行的进程。
2. local CPU包含在进程的cpus_allowed bitmask。
3. 至少一个条件满足：
   - The local CPU is idle. If the kernel supports the hyper-threading technology, all logical CPUs in the local physical chip must be idle.
   - The kernel is having trouble in balancing the scheduling domain, because repeated attempts to move processes have failed.
   - The process to be moved is not "cache hot" (it has not recently executed on the remote CPU, so one can assume that no data of the process is included in the hardware cache of the remote CPU).

如果can_migrate_task()返回1，move_tasks()调用pull_task()函数移动候选进程到local runqueue。如果移动的进程优先级比local CPU上的运行进程高，抢占它。

Chapter 8. Memory Management

8.1. Page Frame Management

8.1.1. Page Descriptors

一个页面帧的状态信息保存在一个page descriptor，主要数据结构是page。所有page descriptors保存在mem_map数组中。

page数据结构的2个成员：

_count:page的引用计数，如果等于-1，相应的page frame是空闲的，能够被分配。如果大于等于0.说明正在使用中（0的话，说明在被内存分配器使用着，未确认）。

flags：描述了page frame的状态。

Flag name	Meaning
PG_locked	页面被锁定；例如被一个磁盘I/O操作锁定。
PG_erro	传输到页面的一个I/O错误。

PG_referenced	页面最近被访问过。
PG_uptodate	成功完成一个I/O读操作后置位，除非有I/O错误发生。
PG_dirty	页面被修改过。
PG_lru	页面在active或inactive page list。
PG_active	页面在active page list。
PG_slab	页面包含在slab中。
PG_highmem	页面属于ZONE_HIGHMEM。
PG_checked	被一些文件系统使用，例如:EXT2或EXT3。
PG_reserved	页面被内核代码reserved或该页面不能使用。
PG_writeback	页面正在被writepage方法写入磁盘。
PG_swapcache	页面属于swap cache。
PG_mappedtodisk	页面所有数据等同于磁盘上的blocks。
PG_reclaim	页面已经被标记为被写入了磁盘，为了回收内存。

8.1.2. Non-Uniform Memory Access (NUMA)

Linux2.6支持Non-Uniform Memory Access (NUMA) model，NUMA下，不同CPU访问不同内存位置所需的时间是不同的。系统物理内存被划分为许多nodes。CPU访问同一个node的页面耗时相同，访问不同node的页面耗时不同。

每一个node有一个描述符，数据结构是pg_data_t,所有的node description存储在一个单链表中，链表的第一个元素地址放在pgdat_list变量中。

Type	Name	Description
struct zone [ ]	node_zones	Array of zone descriptors of the node
struct zonelist [ ]	node_zonelists	Array of zonelist data structures used by the page allocator (see the later section "Memory Zones")
int	nr_zones	Number of zones in the node
struct page *	node_mem_map	Array of page descriptors of the node

Type	Name	Description
struct bootmem_data *	bdata	Used in the kernel initialization phase
unsigned long	node_start_pfn	Index of the first page frame in the node
unsigned long	node_present_pages	Size of the memory node, excluding holes (in page frames)
unsigned long	node_spanned_pages	Size of the node, including holes (in page frames)
int	node_id	Identifier of the node
pg_data_t *	pgdat_next	Next item in the memory node list
wait_queue_head_t	kswapd_wait	Wait queue for the kswapd pageout daemon (see the section "Periodic Reclaiming" in Chapter 17)
struct task_struct *	kswapd	Pointer to the process descriptor of the kswapd kernel thread
int	kswapd_max_order	Logarithmic size of free blocks to be created by kswapd

通常来说，X86都是Uniform Memory Access model，为了可移植性，同样可以使用NUMA，不过此时只有1个node。

8.1.3. Memory Zones

真实的计算机体系有硬件限制，限制了page frame的使用。Linux内核必须应对2种X86的硬件限制：

1. 老式ISA总线上的DMA限制，它们只能寻址RAM的低16MB。
2. 现代32-bit计算机有大量RAM，CPU不能直接寻址所有物理地址，因为线性地址空间太小了。

因此，Linux把物理内存划分为3个zones ：

1. ZONE_DMA: Contains page frames of memory below 16 MB

2. ZONE_NORMAL: Contains page frames of memory at and above 16 MB and below 896 MB

3. ZONE_HIGHMEM: Contains page frames of memory at and above 896 MB

ZONE_DMA和ZONE_NORMAL包含着”normal”page frame，能够被内核通过线性地址映射直接访问。ZONE_HIGHMEM不能被内核直接用过线性地址访问。ZONE_HIGHMEM zone在64bit体系结构中通常为空。

每一个内存zone有数据结构：zone。

每一个page discription被链入内存node和内存node中的zone。为了节约空间，这些links没有用指针实现；而是被编码进了flags的高位。事实上，flags中可用的为很少，但足够编码page frame所属node和所属zone。

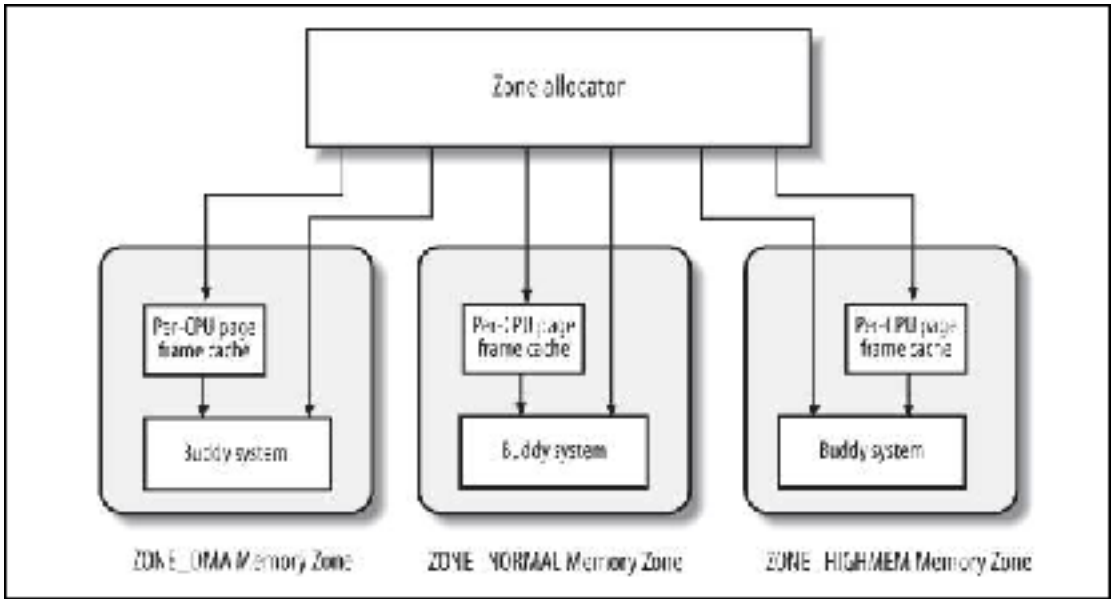
8.1.4. The Pool of Reserved Page Frames

内存分配当遇见内存不够时，可以回收一些内存再分配，但如果是不可阻塞的控制流请求内存，则必须使用atomic memory allocation requests（使用GFP_ATOMIC标志位）。它决不会阻塞：如果没有足够的页面，它会失败。

为了减少atomic memory allocation requests失败的可能性，内核reservs一个pool of page frames对应这种原子内存分配请求，当low-on-memory条件发生时。

reserved memory（KB）总数存储在min_free_kbytes变量中。这个值在内核初始化时被设置，大小依赖于被直接映射的物理内存大小，而且必须在128~65536之间。
zone->pages_min存放了该zone的reserved page frame数目。
zone->pages_min, zone->pages_low, zone->pages_high一起使用在页面回收算法中。

8.1.5. The Zoned Page Frame Allocator



8.1.5.1. Requesting and releasing page frames

Flag used to request page frame :
1. __GFP_DMA: The page frame must belong to the ZONE_DMA memory zone. Equivalent to GFP_DMA
2. __GFP_HIGHMEM: The page frame may belong to the ZONE_HIGHMEM memory zone.
3. __GFP_ZERO: The page frame returned, if any, must be filled with zeros.

__GFP_DMA和__GFP_HIGHMEM被称为zone modfiers；它们指定了内核在寻找free页面时搜索的zone。pg_data_t->node_zonelists是zone descripor数组：for each setting of the zone modifiers, the corresponding list includes the memory zones that could be used to satisfy the memory allocation request in case the original zone is short on page frames. In the 80 x 86 UMA architecture, the fallback zones are the following:

1. 如果__GFP_DMA设置，页面只能从ZONE_DMA分配；
2. 否则，如果__GFP_HIGHMEM没有设置，页面按从ZONE_NORMAL，ZONE_DMA顺序分配。
3. 否则，__GFP_HIGHMEM设置，页面按从ZONE_HIGHMEM,ZONE_NORMAL,ZONE_DMA顺序分配。

8.1.6. Kernel Mappings of High-Memory Page Frames

Linux为了解决32-bit体系中线性地址不够使用的情形，使用了如下方法：

1. 高内存页面只通过alloc_pages()和alloc_page()分配。这两个函数返回相应page descriptor。
2. 线性地址的最高128M被用来临时的映射高物理地址，这样所有高物理地址也能被使用了。

内核使用3种不同的机制来映射高物理地址：permanent kernel mapping，temporary kernel mapping, and noncontiguous memory allocation。

permanent kernel mapping可能会阻塞当前进程；这种情况发生在，当没有free Page Table entries存在时，所以，permanent kernel mapping不能用在中断处理和deferrable function中。相反的是，temporary kernel mapping绝不会阻塞进程，但缺点是在同一时间只有少量临时映射能建立。

8.1.6.1. Permanent kernel mappings

permanent kernel mapping允许内核建立长时间存在的高物理内存映射，它使用的是在master kernel page tables中的一个指定的Page Table。pkmap_page_table变量存放了这个Page Table地址。这个Page Table有512或1024个项，根据是否使用PAE。那么，内核一次最多能访问2或4MB高物理地址。

The Page Table maps the linear addresses starting from PKMAP_BASE. The pkmap_count array includes LAST_PKMAP counters, one for each entry of the pkmap_page_table Page Table. We distinguish three cases:

1. The counter is 0:The corresponding Page Table entry does not map any high-memory page frame and is usable.
2. The counter is 1:The corresponding Page Table entry does not map any high-memory page frame, but it cannot be used because the corresponding TLB entry has not been flushed since its last usage.
3. The counter is n (greater than 1):The corresponding Page Table entry maps a high-memory page frame, which is used by exactly n - 1 kernel components.

kmap()函数用来建立一个permanent kernel mapping。kunmap()用来撤销一个permanent kernel mapping。

8.1.6.2. Temporary kernel mappings

它绝对不会被阻塞，可以使用在中断上下文中。每一个高地址的page frame能通过一个个window（一个内核地址空间名字）映射，一个Page Table项保留用于这个目的。用来做temporary mapping的windows数目很小。

每个CPU有13个windows，主要数据结构为enum km_type。内核必须确保同一个window不会被2个内核控制流同时使用。因此，每个km_type成员的命名，包含了内核组件名。最后一个符号KM_TYPE_NR代表的是windows的总数。因此，每个CPU看到的

temporary kernel mapping都不相同，建立这个映射，修改的应该是master kernel page tables。

km_type中的每一个符号，除了最后一个，是fix-mapped linear address的index。enum fixed_addresses数据结构包含了符号FIX_KMAP_BEGIN 和FIX_KMAP_END，这两个符号指定了temporary mapping对应的fix-mapped linear address。The kernel initializes the kmap_pte variable with the address of the Page Table entry corresponding to the fix_to_virt(FIX_KMAP_BEGIN) linear address.

内核调用kmap_atomic()函数建立temporary mapping。kunmap_atomic()用来撤销映射。

```
void * kmap_atomic(struct page * page, enum km_type type)
{
    enum fixed_addresses idx;
    unsigned long vaddr;
    current_thread_info( )->preempt_count++;
    if (!PageHighMem(page))
        return page_address(page);
    idx = type + KM_TYPE_NR * smp_processor_id( );
    vaddr = fix_to_virt(FIX_KMAP_BEGIN + idx);
    set_pte(kmap_pte-idx, mk_pte(page, 0x063));
    _ _flush_tlb_single(vaddr);
    return (void *) vaddr;
}
```

highmem所占用的虚拟内存分成两个部分,一个是FixMap部分为NR_CPUS个cpu预留了线性地址，共8k的虚存用于kmap_atomic,另外一部分从PKMAP_BASE (0xfe000000UL)开始,预留了4M的虚存.总共能映射1024+2个高端内存页面.

8.1.7. The Buddy System Algorithm

8.1.7.1. Data structures

Linux2.6对每个zone使用一个不同的buddy系统。X86中有3个buddy系统。Buddy系统的主要数据结构：

1. mem_map数组,准确的说，每个zone与mem_map的一个子集相关联。每个zone->zone_mem_map和zone->size确定了各自的范围。
2. free_area类型的数组，11个元素。每个元素对应一个组大小。这个数组在zone->free_area中。Page->lru链入这个队列头。

free_area中链接着的page descriptor中的private预，记录着这个块的大小k。通过这个域，buddy系统就能确定是否能发生合并。

8.1.7.2. Allocating a block

__rmqueue()用来寻找需要的块。

如果找到的块curr_order大于请求的order,一个循环不断执行。

```
size = 1 << curr_order;
while (curr_order > order) {
   area--;
   curr_order--;
   size >>= 1;
   buddy = page + size;
```

```
    /* insert buddy as first element in the list */
    list_add(&buddy->lru, &area->free_list);
    area->nr_free++;
    buddy->private = curr_order;
    SetPagePrivate(buddy);
}
return page;
```

8.1.7.3. Freeing a block

__free_pages_bulk()实现了释放页面的buddy系统。
函数循环10-order次，每次都尽可能的合并buddy块。从小块向大块合并。

```
while (order < 10) {
    buddy_idx = page_idx ^ (1 << order);
    buddy = base + buddy_idx;
    if (!page_is_buddy(buddy, order))
        break;
    list_del(&buddy->lru);
    zone->free_area[order].nr_free--;
    ClearPagePrivate(buddy);
    buddy->private = 0;
    page_idx &= buddy_idx;
    order++;
}
```

8.1.8. The Per-CPU Page Frame Cache

内核的每个memory zone定义了一个per-CPU page frame cache。每个per-CPU cache包含了一些预先分配的page frame对应单个page请求。

对每个zone的每个CPU有两个cache:
hot cache:其中的page很有可能包含在CPU的硬件cache中。如果内核或用户进程在分配一个页面后马上要写数据，那么使用hot cache将会有很作用。
cold cache:如果是DMA操作，那么使用cold cache中的页面将不会修改硬件cache，也就保护了hot cache在硬件cache中的数据。

zone->pageset是一个per_cpu_pageset数据结构的数组，共NR_CPUS个元素。每个元素中有两个per_cpu_pages描述符，分别对应hot cache和cold cache。
内核会监视这两个cache,补充page或释放page。

8.1.8.1. Allocating page frames through the per-CPU page frame caches

buffered_rmqueue()函数用来在给定zone中分配page frame。它利用了per-CPU page cache 来处理单个页面请求。这个函数在cache中page数量少于阈值时会从buddy分配一些page来。如果请求的order大于1，那么从buddy系统中请求。

可见，per-CPU page cache是建立在buddy系统之上的。

8.1.8.2. Releasing page frames to the per-CPU page frame caches

8.1.9. The Zone Allocator

zone allocator必须满足下列要求:

1. 它必须保护the pool of reserved page frames。
2. 它必须触发page回收算法。

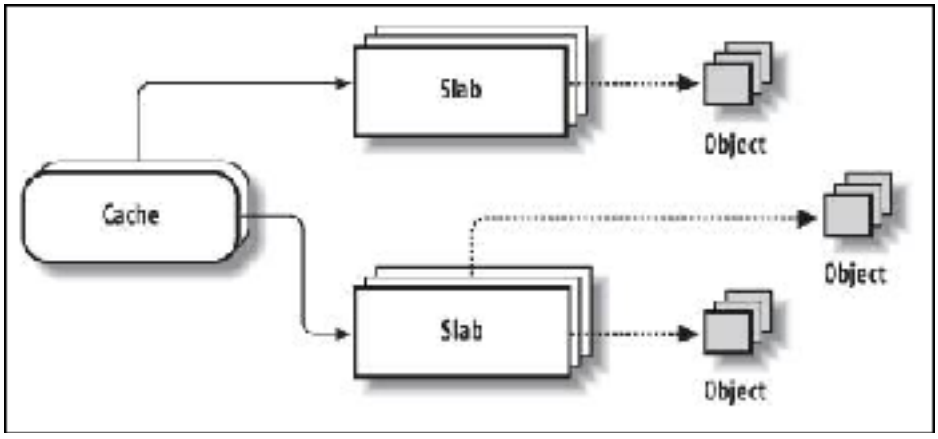
3个参数:

1. gfp_mask
2. order
3. zonelist: 指定了zonelist数据结构，包含了内存分配的管理区优先顺序。

__alloc_pages()按照zonelist指定的顺序搜索每个zone。对每一个zone，函数对比空闲页面的数量和一个阈值，这个阈值依赖于flags，当前进程的类型，zone被检测的次数。如果空闲页面很稀缺了，每个zone将被搜索很多次，每次都会降低阈值。

8.2. Memory Area Management

8.2.1. The Slab Allocator



8.2.2. Cache Descriptor

Cache descriptor主要数据结构是kmem_cache_t。其中的struct kmem_list3 list将连接3个slab队列。

8.2.3. Slab Descriptor

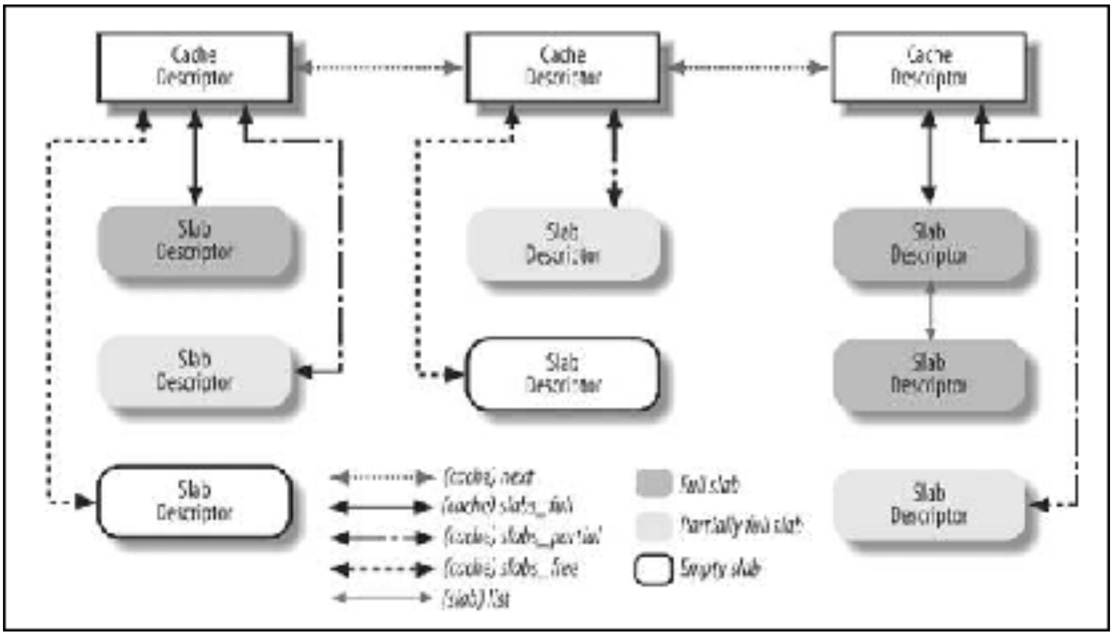
Slab descriptor的主要数据结构slab:

1. list:链入kmem_cache_t->list中slabs_full, slabs_partial, or slabs_free之一。
2. colouroff:slab中第一个对象的偏移。
3. s_mem:slab中第一个对象的地址。
4. inuse:slab中正在使用的对象个数。
5. free:slab中下一个可用对象的index。或者是BUFCTL_END代表没有可用对象。

Slab descriptor可以存在两个地方:

1. External slab descriptor:存在slab外面。
2. Internal slab descriptor:存在slab内部，分配给slab的第一个page的开始

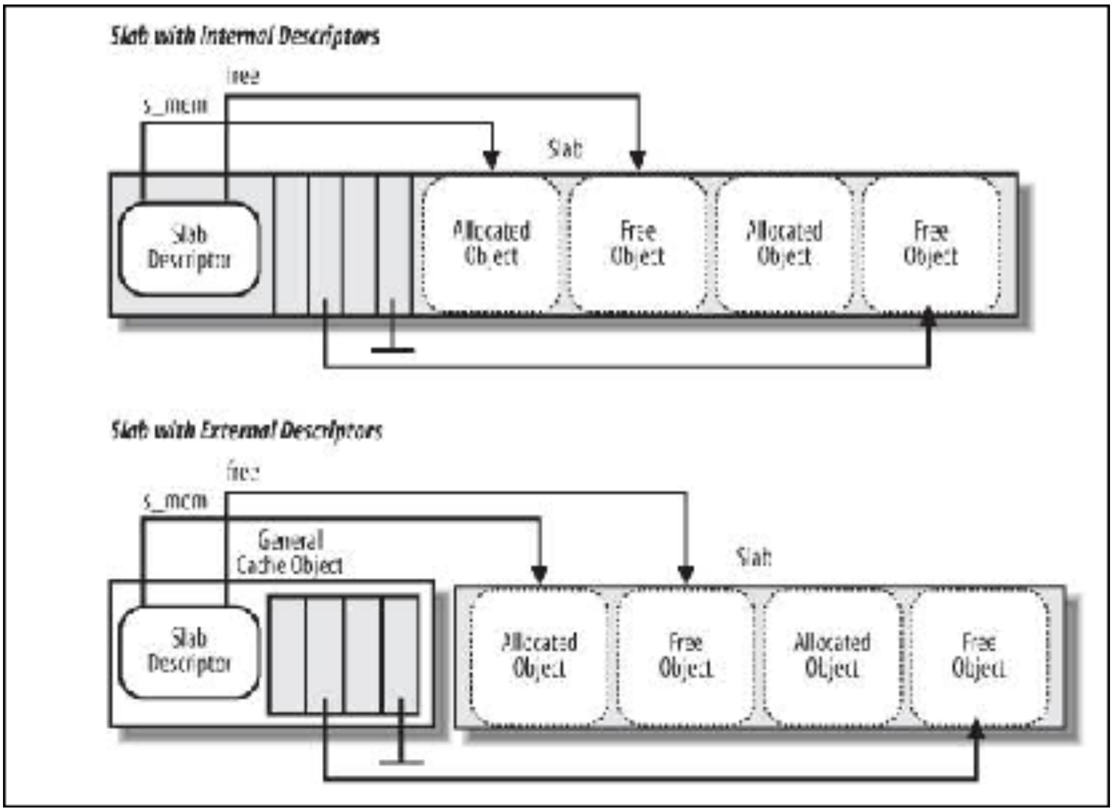
Slab分配器当对象大小小于512KB时，或内部碎片留有足够空间给slab descriptor时，slab descriptor会放在slab内部,此时kmem_cache_t->flags 的CFLGS_OFF_SLAB置0，否则置1。



8.2.8. Object Descriptor

主要数据结构kmem_bufctl_t. Object descriptors 存放在一个数组中，数组放在相应slab descriptor后。有两种存放方式：

1. External object descriptors:存在通用cache，地址在kmem_cache_t->slabp_cache中，通用cache中存放object descriptor的空间大小取决于slab中的object数目。
2. Internal object descriptors:存放在描述的对象前面。



数组中第一个object descriptor描述了slab中第一个object，以此类推。一个object decriptor是一个unsigned short，仅当这个object为free才有效。它包

含了下一个free object。因此在slab中实现了一个简单的链表。最后一个free object的descriptor存的是BUFCTL_END(0xffff);

8.2.9. Aligning Objects in Memory

Object在slab allocator中是在内存中对齐的。也就是说，它在内存中的物理地址是某个常数的倍数，通常是2的乘方。这个常数也叫alignment factor。

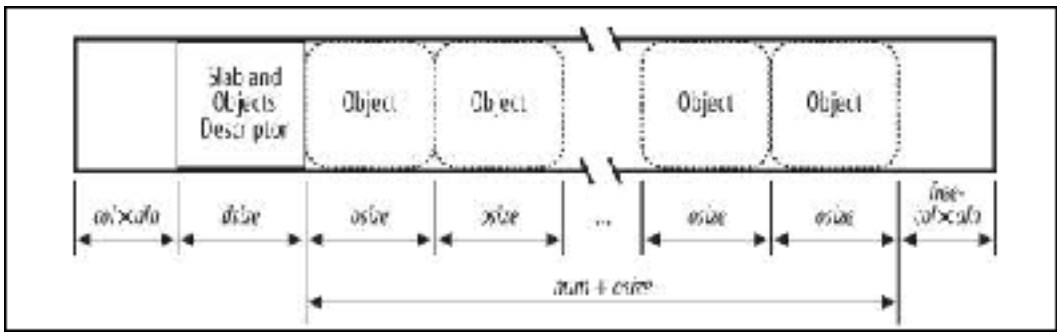
Slab allocator的最大的alignment factor是4096。一个页面大小。这意味着object能被对齐在线性地址或物理地址。无论哪种情况，仅仅地址的低12位可以被对齐所改变。

当我们创建一个新的slab cache，可以指定object对齐L1硬件cache，为了完成这个，内核设置了SLAB_HWCACHE_ALIGN cache descriptor标志，
kmem_cache_create()函数如下处理：

1. 如果object大小大于半个 cache line，那么它将在内存中对齐成L1_CACHE_BYTES的乘方。
2. 否则，object大小变成L1_CACHE_BYTES的约数大小。这确保小object不会跨越2个cache line。

Slab allocator在这里是用空间换时间，人工增加object大小，这就导致了内部碎片。

8.2.10. Slab Coloring



8.2.11. Local Caches of Free Slab Objects

Linux2.6实现对SMPslab allocator，为了减小spin lock的竞争，以及更好的利用硬件cache，每个slab cache包含了一个per-CPU数据结构，由一个很小的数组组成，其成员指向freed object，这被称做salb local cache。绝大多数分配和释放slab object仅影响local cache。

kmem_cache_t->array是一个指向array_cache数据结构的指针数组，有NR_CPUS个元素。每个array_cache数据结构是一个local cache of free object的描述符。

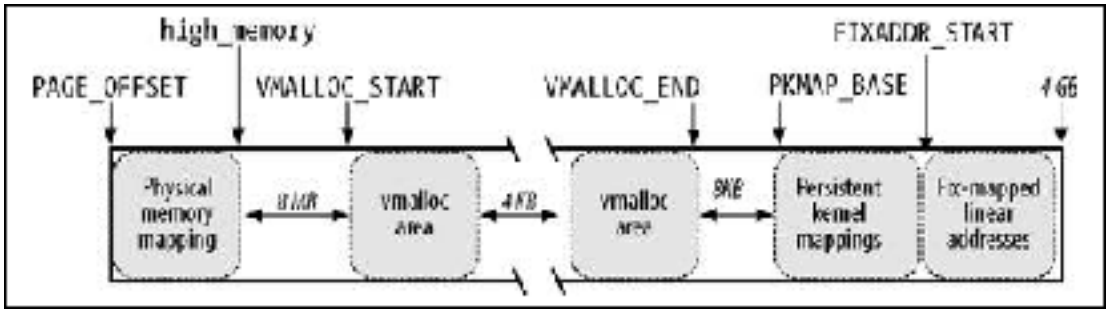
8.2.12. Allocating a Slab Object

8.3. Noncontiguous Memory Area Management

对那些访问不是很频繁的请求，使用把不连续的page frame映射到连续的线性地址,称为noncontiguous memory areas，它的优点是减少了外部碎片，缺点是需要修改内核页面表.这也提供了一种使用高端地址的方法。

8.3.1. Linear Addresses of Noncontiguous Memory Areas

直接映射的物理地址结束处的线性地址保存在high_memory变量中。
从PKMAP_BASE开始是用于高物理地址的persistent mapping的线性地址。
剩下的线性地址是非连续的内存区域。一个8MB(VMALLOC_OFFSET)间隔，用来抓住越界访问。同样，在不同的noncontiguous memory areas之间有4KB的间隔。



8.3.2. Descriptors of Noncontiguous Memory Areas

每个noncontiguous memory area有一个数据结构vm_struct:
addr:区域的起始线性地址。
size:区域大小加上4096(包括了4KB间隔)。
flags:noncontiguous memory area内存映射的类型。
pages:指向一个page descriptor数组,数组大小为nr_pages。
nr_pages:区域里的pages数目。
phys_addr:除非区域被创建为硬件设备的I/O共享内存，否则设为0。
next: 指向下一个vm_struct。

vm_struct组成了一个链表，第一个元素放在vmlist变量中。flags指明了内存区域映射的类型: VM_ALLOC对应通过vmlloc()分配的页面，VM_MAP对应用过vmap()分配的页面，VM_IOREMAP对应于通过ioremap()映射的页面。

get_vm_area()函数在VMALLOC_START and VMALLOC_END寻找一个空的线性地址范围。

1. 调用kmalloc()分配一个vm_struct结构。结构较小，使用的是slab机制。
2. 得到vmlist_lock锁，它是个全局变量。从vmlist开始变历整个vm_struct链表。查找是否有size+4096的线性地址空间。
3. 如果有这么个地址空间，返回the noncontiguous memory area的初始地址。
4. 否则释放资源，返回NULL。

8.3.3. Allocating a Noncontiguous Memory Area

最重要的是map_vm_area()函数:
利用pgd_offset_k宏得到master kernel Page Global Directory中相关于要建立映射区域的初始地址的页表。然后修改master kernel Page Global Directory建立映射。

注意到，当前进程的Page Table并没有被map_vm_area()修改。所以，当一个进程在内核访问noncontiguous memory area,一个Page Fault发生。Page Fault处理函数检查master kernel Page Table中的相应地址。如果发现有非NULL项，就把值复制到进程的Page Table中来，然后恢复进程,(异常处理会使进程的PGD,PMD,PTE指向master kernel Page Table中相应部分,未确定)。

vmap()函数映射已经映射到noncontiguous memory area中的page frame：这个函数接收一个人page descriptor指针数组的参数，调用get_vm_area()得到一个新的vm_struct，然后使用map_vm_area()建立映射。

8.3.4. Releasing a Noncontiguous Memory Area

最关键的是unmap_vm_area()函数，它修改了master kernel Page Global Directory,把相应表项置为0。
和vmalloc()相似，并没有改变进程的Page Table，但实际上，引起Page Fault的进程的pte指向的是master kernel Page Global 相应部分，而此时被清0了，就会opps.

Chapter 9. Process Address Space

__get_free_pages() or alloc_pages()从zone page frame分配页面。
kmem_cache_alloc() or kmalloc() 从slab分配页面。
vmalloc() or vmalloc_32()分配noncontiguous memory area。

9.2. The Memory Descriptor

所有关于进程地址空间的信息都保存在memory descriptor中，其数据结构类型为mm_struct。它是process descriptor的mm成员。mm_struct的主要成员：

1. mmap:指向一个memory region object链表的头。
2. mm_rb:指向一棵红黑树的根，用来管理memory region objects。
3. mmap_cache:指向最后一次引用的memory region object。
4. get_unmapped_area:在进行地址空间寻找一个可用的线性地址间隔。
5. umnmap_area:当释放一个线性地址间隔时调用。
6. mmap_base:标识了第一个分配的memanonymous memory region or file memory mapping的线性地址。
7. free_area_cache:内核在进程地址空间寻找线性地址空间时起始寻找地址。
8. pgd:指向Page Global Directory。
9. mm_users:二级使用计数器。代表共享mm_struct的轻量级进程的数目。所有的使用mm_users的"用户"在mm_count中只算一个。
10. mm_count:主使用计数器。
11. mmlist:指向下邻近的memory descriptor。
12. start_code:可执行代码的初始地址。
13. end_code: 可执行代码的结束地址。
14. start_data:数据段初始地址。
15. end_data:数据段结束地址。
16. brk:当前heap结束地址。
17. rss:分配给进程的page frames数量。

9.2.1. Memory Descriptor of Kernel Threads

为了避免TLB和cache刷新，一个内核线程使用上一次运行的常规进程的Page Tables。

task_struct->mm指向进程所拥有的memory descriptor，而task_struct->active_mm指向进程执行时使用的memory descriptor。对常规进程来说，这两个成员是相等的。内核线程，没有自己的memory descriptor，所以mm为NULL。当一个内核线程被选择来使用时，它的active_mm被初始化为前一个运行的进程的active_mm。

9.3. Memory Regions

Linux实现了一个memory region，主要数据结构是vm_are_struct，其主要成员是：

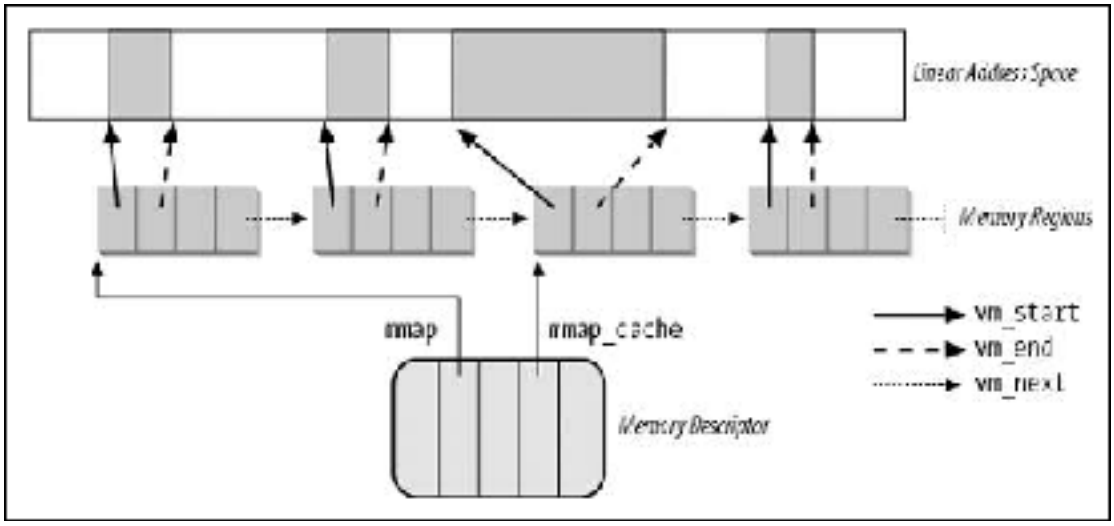
1. vm_mm：指向拥有该region的memory descriptor。
2. vm_start：region的初始地址，这个地址包含在region内。
3. vm_end:region后的第一个地址，这个地址不包含在region内。
4. vm_next：进程中下一个region。
5. vm_ops：memory region的方法。
6. vm_pgoff:在mapped file中的偏移。对anonymous pages，它要么是0要么是vm_start/PAGE_SIZE。
7. vm_file:指向一个被映射的文件的file对象。

Memory regions不会发生重叠。内核当分配新region时，会尽可能的合并相连的region,前提是访问权限匹配。

vm_area_struct->vm_ops包含的四种方法：

1. open：当memory region被加入到regions集合时调用。
2. close：当memory region被移除时调用。
3. nopage:当Page Fault发生，进程访问的页面不在RAM中，它的线性地址属于这个memory region时调用。
4. populate:用来设置memory region的线性地址相应的page table entries，主要用在non-linear file memory mapping。

9.3.1. Memory Region Data Structures



Linux既用线性链表组织memory region又用红黑树来组织memory region。

9.3.2. Memory Region Access Rights

与page有关的3中flags:

1. 存储在Page Table entry中的Read/Write, Present, or User/Supervisor，它们被X86硬件用来检测寻址是否能执行。
2. 每个page中的flags，被Linux用在多种用途中。
3. 存储在vm_area_struct中的vm_flags，一些flags用来提供region内所有pages的信息，例如what they contain and what rights the process has to access each page。其它flags描述了region自己，例如如何增长。

Memory region内的page访问权限可以被任意的组合，权限的设定必须复制到相应的Page Table entries,这样就能用硬件完成权限的检测。换句话说，page访问权限代表了何种访问会产生Page Fault 异常。

Page Table flags的初始值存放在vm_area_struct->vm_page_prot中。当增加page时，内核把相应的Page Table entry设置为vm_page_prot的值。

然而，把memory region的访问权限设置给page的保护bits不是直接进行的，这是因为:

1. 在某些时候，一个page的访问应该产生Page Fault异常，即使它的存储权限被region的vm_flags保证了。例如，内核希望映射属于两个不用进程的，可写的私有页面到同一个page frame,那么，任何一个进程试图修改页面都将引发页面异常。
2. X86有连个保护位，其中User/Supervisor在memory region的页面必须置位，因为region里的页面总是能被用户态进程访问。

为了正确的通过Copy On Write技术推迟真正的分配page frame，都使page frame变成写保护的。

页面权限可以由Read, Write, Execute, 和 Share access rights产生16种组合，其规则如下:

1. 如果有Write和Share权限，Read/Write置位。
2. 如果有Read或Execute权限，但没有Write或Share 权限，Read/Write清0.
3. 如果NX bit被支持，并且page没有Execute权限，NX置位。

4. 如果page没有任何访问权限，Present被清0，那么每个访问就会产生Page Fault异常。而且，为了与真正的page-not-present区别，Linux还将Page size置位。

9.3.3. Memory Region Handling

9.3.3.1. Finding the closest region to a given address: find_vma( )

find_vma()函数定位vm_end大于addr的第一个memory region,并且返回region描述符地址。如果找不到，返回NULL。注意到，被find_vma()找到的region不一定包含了addr。

find_vma_prev()功能类似与find_vma()，不同的是，它还把找到的memory region的前一个region的地址写入pprev参数指定的地址。

find_vma_prepare()…暂缺。。。

9.3.3.2. Finding a region that overlaps a given interval: find_vma_intersection()

find_vma_intersection()函数找出包含一个地址范围的vm_area_struct。

9.3.3.3. Finding a free interval: get_unmapped_area()

get_unmapped_area()函数寻找进程地址空间，寻找一个可用线性地址间隔。len参数指定了间隔的长度，一个非NULL addr参数指定了搜索起始地址。

9.3.3.4. Inserting a region in the memory descriptor list: insert_vm_struct( )

insert_vm_struct()把一个vm_area_struct structure 插入 memory region object list和red-black tree of a memory descriptor.

9.3.4. Allocating a Linear Address Interval

do_mmap()函数为current进程创建并且初始化一个新的memory region，在成功分配region后，有可能的话，要进行region的合并。
函数使用下列参数:

1. file和offset: 文件对象指针file和文件偏移offset用在新memory region将映射一个文件到内存。
2. addr:这个线性地址指定了搜索一个空间隔时的起始地址。
3. len:线性地址间隔的长度。
4. prot:指定了包含在memory region内的page的访问权限。可能的flags有PROT_READ, PROT_WRITE, PROT_EXEC, and PROT_NONE。头三个权限等价于VM_READ, VM_WRITE, and VM_EXEC flags。PROT_NONE 指示进程没有访问权限。
5. flags指定了剩下的memory region flags:

接着执行do_mmap_pgoff()来实现anonymous memory regions:

1. 检查参数合法性和请求是否能满足:
2. 调用get_unmapped_area()得到一个线性地址间隔给新的region。
3. 把prot和flags合并起来，计算出新memory region的flags。
4. 调用find_vma_prepare()定位在新间隔之前一个memory region对象地址，存在vma中，和new region在红黑树中的位置。它还检查memory region是否是新间隔有重叠。
5. 检测插入新memory region后是否会导致进程地址空间超过阈值。

6. 如果MAP_NORESERVE flag没有设置，新memory region含有私有的可写页面，并且没有足够的空page frame，那么返回-ENOMEM。
7. 如果新间隔是私有的，没有映射到磁盘上的一个文件。调用vma_merge()检测是否可以把vma之前找到的memory region扩充来包含该新间隔。当然，它们必须有相同的flags。
8. 分配一个vm_area_struct数据结构给新memory region。
9. 初始化新memory region对象。
10. 如果MAP_SHARED置位（并且没有映射到磁盘文件），那么region是一个共享匿名region：调用shmem_zero_setup()初始化它。共享匿名region主要用在进程间通信。
11. 插入红黑树。
12. 递增mm_struct->total_vm。
13. 如果VM_LOCKED置位，调用make_pages_present分配所有的页面并锁在内存里。页面是一个一个分配的。
14. 结束并返回新memory region的线性地址。

9.3.5. Releasing a Linear Address Interval

使用do_munmap()删除一个线性地址间隔。注意，删除的间隔不一定是一个完整的memory region；它可能包含在一个region里，也可能跨越了多个region。

9.3.5.1. The do_munmap( ) function

这个函数主要有两个过程：第一步，搜索memory region链表，把所有包含在线性地址间隔里的region脱链。第二步，更新Page Tables移除第一步中的region标识符。详细步骤：

1. 合法性检测。
2. 定位在线性地址间隔后的第一个memort region，存放在mpnt中。
   mpnt = find_vma_prev(mm, start, &prev);
3. 如果没有这样的memory region或region没有覆盖线性地址间隔，那么返回，因为没有region在间隔里。
4. 如果线性地址间隔起点在mpnt memory region中，调用split_vma()分割mpnt memory region成两个较小的region ：一个在间隔外，一个在间隔内。
5. 如果线性地址间隔end地址在一个人memory region中，再次调用split_vma()分割最后一个与线性地址间隔重叠的memory region。
6. 更新mpnt使之指向线性地址间隔中的第一个memory region。
7. 调用detach_vmas_to_be_unmapped()来移除包含在线性地址间隔内的memory region。
8. 得到mm->page_table_lock spin lock。
9. 调用unmap_region清除覆盖线性地址间隔的Page Table entries和释放相应的page frames。
10. 释放mm->page_table_lock spin lock。
11. 释放memory region描述符。
    执行了mm->umap_area()，mm->close()。
12. 返回。

9.4. Page Fault Exception Handler

do_page_fault()是Page Fault异常的处理函数。两个参数：
regs:指向异常发生时的寄存器值保存在内核栈中的地址。

error_code:3-bit，如果bit0清零，异常是因为page is not present。如果bit0置位，异常是非法的访问权限导致。如果bit1清零，异常由read或excute访问引起，如果置位，异常由write引起。如果bit2清零，异常发生在内核态，否则发生在用户态。

do_page_fault()第一步操作是读取导致异常的线性地址，这个地址在cr2寄存器中。

如果异常线性地址是在内核空间，处理程序检测是否是内核尝试访问不存在的page frame，跳转到vmalloc_fault。

接着，处理程序检测该异常是否发生在内核正在执行关键程序或在执行内核线程。

```
if (in_atomic( ) || !tsk->mm)
    goto bad_area_nosemaphore;
```

in_atomic()返回1，当异常发生在如下环境时:
内核在执行中断处理或deferrable function。
内核在执行关闭抢占的临界区。
正在执行关键程序或在执行内核线程就意味这错误发生,因为内核线程绝不会使用TASK_SIZE以下的线性地址，中断处理，defferable function,临界区都不会使用TASK_SIZE以下的线性地址，因为这可能会阻塞当前进程。

异常处理函数检测线性地址是否发生在进程空间，为了检查线性地址是否在memory region中，必须获取进程的mmap_sem read/write semphone:

```
if (!down_read_trylock(&tsk->mm>mmap_sem)) {
    if ((error_code & 4) == 0 &&
        !search_exception_table(regs->eip))
        goto bad_area_nosemaphore;
    down_read(&tsk->mm->mmap_sem);
    }
```

当前进程在Page Fault发生时可能得不到mmap_sem，使用down_read_trylock()防止死锁。如果semphone是关闭的，Page Fault发生在内核态，那么do_page_fault()判断whether the exception occurred while using some linear address that has been passed to the kernel as a parameter of a system call。如果引起异常的线性地址是系统调用的参数，那么do_page_fault()明白semaphore被另一个进程获取了，因为每个系统调用服务程序都很小心的避免获取semaphore，因此do_page_fault()将等待semaphore的释放。否则，那么Page Fault是内核bug或严重的硬件故障。

如果找不到包含引起异常的线性地址的region，那么跳转到bad_area；否则进入good_area。

```
vma = find_vma(tsk->mm, address);
if (!vma)
    goto bad_area;
if (vma->vm_start <= address)
    goto good_area;
```

如果两个if都不满足，那么要进一步检测，看页面异常是否是由push或pusha指令产生，这种情况下就要扩充用户栈。

关于用户栈映射到memory region的问题：用户栈region是向下扩展的，vm_end地址不变。vm_tart可以减少。VM_GROWSDOWN置位。Region大小是4KB对齐的，而用户栈

的大小是任意的。分配给region的page frames绝对不会释放，除非region被删除。而且，vm_start只能减少，绝对不会增加，即使进程执行了一系列pop指令，region大小依然不改变。

push操作会引用一个在用户栈region外的地址，这种异常不是编程错误，需要单独处理。

```
if (!(vma->vm_flags & VM_GROWSDOWN))
      goto bad_area;
if (error_code & 4      /* User Mode */
        && address + 32 < regs->esp)
      goto bad_area;
if (expand_stack(vma, address))
      goto bad_area;
   goto good_area;
```

因为一条关于栈的汇编指令最多减少esp 32个字节。所以调用expand_stack()函数来检测进程是否允许扩展栈和地址空间。注意到，对于发生在内核态的异常，上述代码一定会执行expand_stack()。

9.4.1. Handling a Faulty Address Outside the Address Space

如果线性地址不属于进程地址空间，do_page_fault()执行bad_area代码，如果是发生在用户态，口发送品尼高SIGSEGV给当前进程。

如果异常发生在内核态，仍有两种选择：
发生在一些线性地址作为系统调用参数传给内核。
异常是真正的内核bug。

这两种情况的区别在：

```
no_context:
if ((fixup = search_exception_table(regs->eip)) != 0) {
   regs->eip = fixup;
   return;
}
```

第一种情况是跳转到”fixup code”，典型的动作是发送一个SIGSEGV信号给当前进程，或结束系统调用。
第二种情况，做一些调试工作。

9.4.2. Handling a Faulty Address Inside the Address Space

如果线性地址在用户进程空间，do_page_fault()执行good_area代码：

```
good_area:
info.si_code = SEGV_ACCERR;
write = 0;
if (error_code & 2) { /* write access */
    if (!(vma->vm_flags & VM_WRITE))
            goto bad_area;
    write++;
    } else                /* read access */
     if ((error_code & 1) || !(vma->vm_flags & (VM_READ |
VM_EXEC)))
       goto bad_area;
```

如果异常是由写访问造成的，函数检查memory region是否可写，如果不可写，跳转到bad_area，否则，把write局部变量设为1。

如果异常是由读或执行访问造成的，函数首先检查页面是否在RAM中。如果发现是因为用户态下访问特全page frame，函数跳转到bad_area，如果页面不再RAM中，函数检测memory region是否可读或可执行。

如果memory region访问权限匹配上了访问的页面的权限，handle_mm_fault()调用，来分配一个新page frame:

```
survive:
ret = handle_mm_fault(tsk->mm, vma, address, write);
if (ret == VM_FAULT_MINOR || ret == VM_FAULT_MAJOR) {
if (ret == VM_FAULT_MINOR) tsk->min_flt++; else tsk->maj_flt++;
 up_read(&tsk->mm->mmap_sem);
 return;
}
```

handle_mm_fault()如果成功的分配了一个新page frame，则返回VM_FAULT_MINOR or VM_FAULT_MAJOR。VM_FAULT_MINOR指示Page Fault被处理了，而且没有阻塞当前进程，这叫做minor fault。VM_FAULT_MAJOR 指示了Page Fault强迫了当前进程进入睡眠(最有可能是因为在从磁盘传输数据给分配的page frame)，会阻塞当前进程的Page Fault叫majaor fault。handle_mm_fault()还会返回VM_FAULT_OOM (没有足够的内存) or VM_FAULT_SIGBUS (其他错误).

如果handle_mm_fault()不能分配新的page frame，它返回VM_FAULT_OOM，内核通常会杀死当前进程。而且，如果当前进程是init进程，那么它将被放在进程队列尾部，然后产生调度。一旦init恢复运行，handle_mm_fault()将被再次执行。

handle_mm_fault()中:
handle_pte_fault()函数调用来检测对应于引起异常的线性地址对应的PTE，决定如何分配一个新的page frame给进程:

1. 如果被访问的页面不存在，内核分配一个新oage frame并初始化；这叫做demand paging。
2. 如果被访问的页面在内存中，但是被标记为read-only，那么内核将分配新的page frame，并用原来的页面内容初始化新page frame。

9.4.3. Demand Paging

Demand paging代表了一种动态内存分配技术，它推迟了page frame的分配，直到最后一刻，当进程要访问时，会造成Page Fault异常来分配page frame。

页面异常处理函如何初始化分配给进程的页面，有如下情况:

1. 页面没有被访问过并且没有映射到一个文件，或者页面映射了一个文件。内核通过PTE是0可以判断出这种情况。
2. 页面属于一个非线性磁盘文件映射。内核通过判断Present flag清0，和Dirty flag置位判断出这种情况。
3. 页面已经被访问过，但是它的内容被临时放在磁盘上。内核通过判断PTE非0，但Present和Dirty flag清0来识别这种情况。

当page没有被访问过或page线性映射到一个磁盘文件，do_no_page()函数被调用。有两种方法装载页面，依赖与页面是否映射到一个磁盘文件。函数检查vma的nopage方法，它指定了当页面映射着文件时，装载缺失的页面进RAM的方法。所以，可能的情况有：

1. vma->vm_ops->nopage非NULL。这说明memory region映射到了一个磁盘文件，nopage代表装载页面的方法。
2. vma->vm_ops field 或 the vma->vm_ops->nopage是NULL，说明memory region没有映射到文件。这是个**anonymous mapping**。do_no_page()将调用do_anonymous_page()函数得到一个新的page frame。

do_anonymous_page()函数分别处理write和read请求：

```
if (write_access) {
pte_unmap(page_table);
spin_unlock(&mm->page_table_lock);
page = alloc_page(GFP_HIGHUSER | _ _GFP_ZERO);
spin_lock(&mm->page_table_lock);
page_table = pte_offset_map(pmd, addr);
mm->rss++;
entry = maybe_mkwrite(pte_mkdirty(mk_pte(page,
                                                    vma-
>vm_page_prot)), vma);
lru_cache_add_active(page);
SetPageReferenced(page);
set_pte(page_table, entry);
pte_unmap(page_table);
spin_unlock(&mm->page_table_lock);
return VM_FAULT_MINOR;
}
```

先调用pte_unmap宏释放之前temporary kernel mapping映射的Page Table entry，因为alloc_page()会阻塞进程，而如果进程被转移到另一个CPU，那么将访问不了之前temporary kernel mapping映射的内容。

注意到，alloc_page(GFP_HIGHUSER | _ _GFP_ZERO)可见映射到用户空间的page frame都是尽力从高物理地址取得。

这里使用temporary kernel mapping建立PTE映射表，是为了尽力使用高物理地址页面。当一次缺页异常发生时，就先建立temporary kernel mapping，然后建立合适的PTE映射，然后回收temporary kernel mapping的线性地址，又可以用在下一次缺页异常中。

相反的，对读访问，并不是分配给它一个新page frame,然后填充0。而是使PTE直接指向一个公用的zero page，这样就推迟了page frame的分配。zero page是内核初始化时静态分配的，保存在empty_zero_page变量中。

9.4.4. Copy On Write

Copy On Write(COW)技术，在建立子进程时，并不是复制父进程所有页面，而是共享父进程页面。当父进程或子进程尝试去写一个共享页面时，异常发生。此时，内核才真正复制页面，并标记其为可写。原始的页面仍然是只读的：当其它进程尝试写时，内核检查是否写进程是页面的唯一拥有者；在这种情况下，标记page frame 为可写。

当handle_pte_fault()确定Page Fault异常是由于访问在内存中的页面造成的，函数执行下列语句：

```
if (pte_present(entry)) {
    if (write_access) {
```

```
        if (!pte_write(entry))
           return do_wp_page(mm, vma, address, pte, pmd, entry);
        entry = pte_mkdirty(entry);
        }
 entry = pte_mkyoung(entry);
 set_pte(pte, entry);
 flush_tlb_page(vma, address);
 pte_unmap(pte);
 spin_unlock(&mm->page_table_lock);
 return VM_FAULT_MINOR;
}
```

如果页面在内存，且是对写保护页面的写操作造成的异常，那么调用do_wp_page()函数。它将确定是否要使用COW。基本的，函数先读取page->_count的值，如果等于0（单个用户），COW不启用。事实上，检测会更复杂些，因为_count当页面被插入swap cache时和PG_private置位时也会递增。然而，如果COW没有启动的话，page frame会被标记为可写,以后它就不会再被触发Page Fault异常了。

如果页面是被在多个进程间共享的话，就会使用COW，函数会拷贝原有页面的内容到新页面。

因为分配page frame能阻塞进程，函数检查Page Table entry是否被修改过（pte和*page_table含有不同的值）。在这种情况下，新分配的page frame被释放。函数结束。

如果所有事都OK了，新page frame的物理地址最后被写入Page Table entry，相应的TLB寄存器失效。

9.4.5. Handling Noncontiguous Memory Area Accesses

内核在更新noncontiguous memory areas的Page Tables时,更新的仅仅只是master kernel Page Tables。

当一个进程在内核态第一次访问一个noncontiguous memory areas时，当把线性地址转换成物理地址，CPU的内存管理单元遇见了一个null页面，Page Fault发生。Page Fault处理函数发现异常发生在内核态，而且引起生异常的线性地址在TASK_SIZE以上，于是do_page_fault()处理函数检查相应的master kernel Page Tables。

```
    vmalloc_fault:
    asm("movl %%cr3,%0":"=r" (pgd_paddr));
    pgd = pgd_index(address) + (pgd_t *)__va(pgd_paddr);
    pgd_k = init_mm.pgd + pgd_index(address);
    if (!pgd_present(*pgd_k))
        goto no_context;
    pud = pud_offset(pgd, address);
    pud_k = pud_offset(pgd_k, address);
    if (!pud_present(*pud_k))
        goto no_context;
    pmd = pmd_offset(pud, address);
    pmd_k = pmd_offset(pud_k, address);
    if (!pmd_present(*pmd_k))
        goto no_context;
    set_pmd(pmd, *pmd_k);
    pte_k = pte_offset_kernel(pmd_k, address);
    if (!pte_present(*pte_k))
        goto no_context
```

pgd_paddr局部变量装载这当前进程的Page Global Derectory的物理地址(从cr3寄存器得到)。pgd局部变量装载着引起异常的线性地址在当前进程Page Global Derectory中对应的项的线性地址，pgd_k局部变量装载了引起异常的线性地址在master kernel Page Global Directory中对应的项的线性地址。
如果master kernel Page Global Directory对应项为NULL，函数跳转到no_context处的代码。否则，函数查看master kernel Page Upper Directory entry 和master kernel Page Middle Directory entry 。如果其中一个为null，跳转到no_context处代码。否则，拷贝master entry内容到进程的Page Middle Directory中（注意PTE项指向的是master kernel Page Tables中的，所以vfree()只需要删掉master kernel Page Tables中的对应项即可。）

9.5. Creating and Deleting a Process Address Space

9.5.1. Creating a Process Address Space

如果新进程通过clone()系统调用哪个创建，并且CLONE_VM flag置位，copy_mm()简单的使新进程和父进程共享同一个mm_struct。

```
if (clone_flags & CLONE_VM) {
    atomic_inc(&current->mm->mm_users);
    spin_unlock_wait(&current->mm->page_table_lock);
    tsk->mm = current->mm;
    tsk->active_mm = current->mm;
    return 0;
}
```

如果CLONE_VM没有设置，copy_mm()必须创建一个新的地址空间。copy_mm()创建一个新的mm_struct结构，挂在新进程描述符tsk上。把当前进程的current->mm的内容拷贝进tsk->mm中。不过要做一些小小的修改。

```
    tsk->mm = kmem_cache_alloc(mm_cachep, SLAB_KERNEL);
    memcpy(tsk->mm, current->mm, sizeof(*tsk->mm));
    atomic_set(&tsk->mm->mm_users, 1);
    atomic_set(&tsk->mm->mm_count, 1);
    init_rwsem(&tsk->mm->mmap_sem);
    tsk->mm->core_waiters = 0;
    tsk->mm->page_table_lock = SPIN_LOCK_UNLOCKED;
    tsk->mm->ioctx_list_lock = RW_LOCK_UNLOCKED;
    tsk->mm->ioctx_list = NULL;
    tsk->mm->default_kioctx = INIT_KIOCTX(tsk->mm-
>default_kioctx,
                                          *tsk->mm);
    tsk->mm->free_area_cache = (TASK_SIZE/3+0xfff)&0xfffff000;
    tsk->mm->pgd = pgd_alloc(tsk->mm);
    tsk->mm->def_flags = 0;
```

之后会调用dup_mmap()函数复制父进程的memory region和Page Tables:
这个函数把tsk->mm插入memory descriptors的全局链表。并且扫描current->mm->mmap,复制其中的memory region，插入到tsk->mm的相应链表中。
在每插入一个新memory region descriptor，dup_mmap()调用copy_page_range(),在必要时复制Page Table中映射这memory region的项。最后，memory region(VM_SHARED清0，VN_MATWRITE置位)中每一个私有的，可写page frame被标记成对父子进程只读，这样就能被COW机制处理了。

9.5.2. Deleting a Process Address Space

当一个进程结束时，内核调用exit_mm()函数释放进程地址空间：

```
mm_release(tsk, tsk->mm);
if (!(mm = tsk->mm)) /* kernel thread ? */
   return;
down_read(&mm->mmap_sem);
```

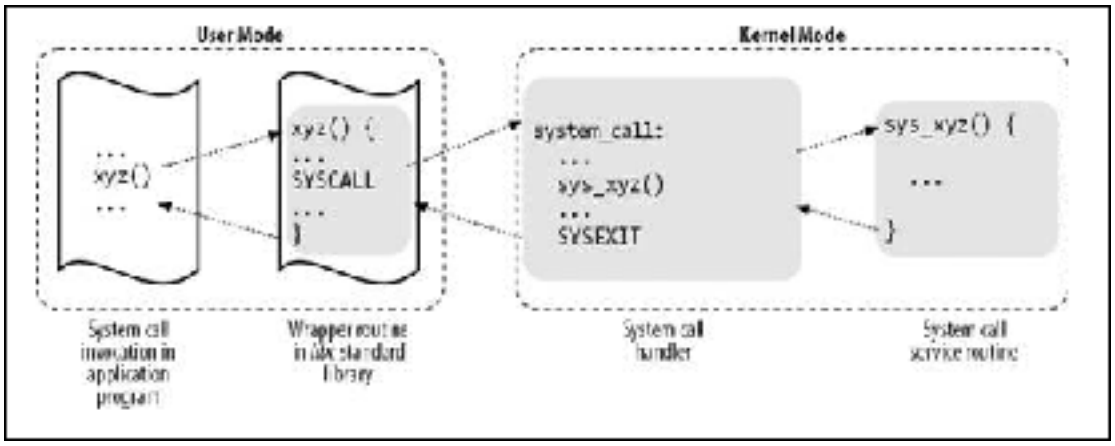
mm_release()函数本质上唤醒了所有睡眠在tsk->vfork_done上的进程。典型的，相应的等待队列为非空，当且仅当退出进程是被vfork()创建的。

如果结束的进程是不是内核线程，exit_mm()函数必须释放memory descriptor和所有相关数据结构。

memory descriptor将被finish_task_switch( )施放，不过这要当进程被有效的从当前CPU移除时才会释放。

Chapter 10. System Calls

10.2. System Call Handler and Service Routines



与每个系统调用相联系的是相应的服务程序，内核使用一个system call dispatch table，它存储在sys_call_table数组中，有NR_syscalls个entries。NR_syscalls代表的使系统调用数目的上限，但不是当前系统调用个数，对没有实现的系统调用，相应entry指向sys_ni_syscall()函数。

10.3. Entering and Exiting a System Call

应用程序使用系统调用有两种方法：

1. 执行int $x80汇编语言，在老版本的Linux内核版本中，这是系统调用的唯一办法。
2. 执行sysenter汇编语言，这条指令被Linux2.6内核支持了。

相似的，内核退出系统调用切换回用户空间，也有两种方法：

1. 执行iret汇编语言。
2. 执行sysexit汇编语言。与sysenter一起使用。

10.3.1. Issuing a System Call via the int $0x80 Instruction

系统调用对应的中断向量号是128(0x80)，内核初始化IDT：

```
set_system_gate(0x80, &system_call);
```

它的gate descriptor是:
1. Segment Selector:__KERNEL_CS。
2. Offset: system_call()的地址。
3. Type:15，指示出它是一个陷阱，相应的响应函数不会关中断。
4. DPL: 3，它使进程能从用户模式穿越陷阱门。
所以，一旦用户使用ibt $0x80指令，CPU将切换到内核模式，开始执行system_call()。

10.3.1.1. The system_call( ) function

system_call()函数从保存系统调用号和保存所有将被使用的CPU寄存器到内核栈中开始（不包括eflags,cs,eip,ss和esp，他们已经被控制单元自动保存）。SAVE_ALL宏，还会将__USER_DS装载进ds和es:

```
system_call:
 pushl %eax
 SAVE_ALL
 movl $0xffffe000, %ebx /* or 0xfffff000 for 4-KB stacks */
 andl %esp, %ebx
```

函数接着把thread_info地址放入%ebx。

对用户传来的系统调用号进行一次合法性检测。如果大于等于system call dispatch table的大小，系统调用处理函数结束。

```
cmpl $NR_syscalls, %eax
jb nobadsys
movl $(-ENOSYS), 24(%esp)
jmp resume_userspace
    nobadsys:
```

最后，特定的服务程序被调用，系统调用号存在%eax中。

```
call *sys_call_table(0, %eax, 4)
```

因为每个entry是4字节，内核把系统调用好乘以4再加上sys_call_table地址，去除服务程序的地址。

10.3.1.2. Exiting from the system call

当系统调用结束，system_call()函数从%eax得到服务程序的返回值，然后存放在栈里eax保存的地方:

```
movl %eax, 24(%esp)
```

这样，用户进程将从%eax中得到系统调用的返回值。

System_call()关闭中断，检查current->thread_info的flag:

```
cli
movl 8(%ebp), %ecx
testw $0xffff, %cx
je restore_all
```

在正式返回用户空间前，还要检测调度请求，virtual-8086模式，未定的信号和single stepping;然后跳转到restore_all重启用户进程的执行。

10.3.2. Issuing a System Call via the sysenter Instruction

sysenter汇编语言利用了3个特别的寄存器，它们装载了下列信息:
1. SYSENTER_CS_MSR:
The Segment Selector of the kernel code segment

2. SYSENTER_EIP_MSR:
The linear address of the kernel entry point
3. SYSENTER_ESP_MSR:
The kernel stack pointer

当sysenter执行时，CPU控制单元:
1. 拷贝SYSENTER_CS_MSR到CS;
2. 拷贝SYSENTER_EIP_MSR到eip;
3. 拷贝SYSENTER_ESP_MSR到esp;
4. SYSENTER_CS_MSR加8，然后装载到ss。
这样，CPU切换到了内核态，并开始执行SYSENTER_EIP_MSR指向的第一条指令。

这3个特殊寄存器被enable_sep_cpu()初始化，每个CPU在系统初始化时都调用该函数一次。这个函数执行下列步骤:
1. 把__KERNEL_CS写入SYSENTER_CS_MSR。
2. 把sysenter_entry()函数地址写入SYSENTER_EIP_MSR寄存器。
3. 计算local TSS末尾的线性地址，把值写入SYSENTER_ESP_MSR寄存器。
当一个系统调用开始时，内核栈为空，esp寄存器指向的是4K或8K内存区的末端，内存区的开始处存放的是当前进程的thread_info结构。用户态程序不能设置esp，因为它不知道内核栈的地址。然而，在进入内核态时，该寄存器必须正确设置。所以，内核把这个寄存器用local CPU的TSS地址。TSS的esp0保存了内核栈地址，在进程切换时会被更新，使之总是指向当前进程的内核栈。系统调用处理函数读取esp寄存器值，计算local TSS的esp0域，装载它的值进入esp寄存器，使之指向内核栈指针。

10.3.2.2. The vsyscall page

libc标准库当且仅当CPU和Linux内核都支持时，才能使用sysenter指令。

兼容性问题已用一个非常巧妙的方法解决了。在系统初始化阶段，sysenter_setup()函数创建了一个page frame，叫做vsyscall页面，包含了一个小的ELF共享对象。当一个进程发出execve()系统调用开始执行ELF程序，在vsyscall 页面里的代码动态的链接到进程地址空间。Vsyscall页面的代码利用了最合适的指令来发出系统调用。

sysenter_setup()函数分配了一个新page frame给vsyscall，把它的线性地址与FIX_VSYSCALL fix-mapped线性地址相联系。然后，函数把如下预先定义好的ELF共享对象拷贝进分配的页面。
1. 如果CPU不支持sysenter，函数在vsyscall页面中包含如下代码:
```
__kernel_vsyscall:
    int $0x80
  ret
```
2. 否则，如果CPU支持sysenter，函数在vsyscall页面中包含如下代码:
```
__kernel_vsyscall:
   pushl %ecx
   pushl %edx
   pushl %ebp
   movl %esp, %ebp
    sysenter
```

如果程序必须调用系统调用，它将调用__kernel_vsyscall()函数。

10.3.2.3. Entering the system call

通过sysenter指令调用系统调用的步骤：

1. 标准库程序把系统调用号装载进%eax，调用__kernel_vsyscall()函数。
2. __kernel_vsyscall()把ebp,edx,ecx保存在用户栈，把用户栈指针拷贝到ebp，然后执行sysenter指令。
3. CPU切换到内核态，内核执行sysenter_entry()函数(SYSENTER_EIP_MSR指向它)。
4. sysenter_entry()汇编指令执行下列步骤：
   - 建立内核栈指针：
     ```
     movl -508(%esp), %esp
     ```
     开始时，esp指向的是local TSS的末尾地址，所以，指令要装载TSS中的esp0进入%esp，这样%esp才真正的指向了当前进程的内核栈。
   - 开中断。
   - 在内核栈中保存用户数据段Segment Selector，当前用户栈指针，eflags寄存器值，用户代码段Segment Selector，从系统调用的返回地址。
   - 恢复%ebp在用户态时的值。
     ```
     movl (%ebp), %ebp
     ```
   - 调用系统调用处理程序执行等同于从system_call标签开始的指令序列。

10.3.2.4. Exiting from the system call

sysenter_entry()函数的返回本质上执行的操作和system_call()相同。首先，把系统调用服务程序的返回码存放在内核栈中，对应用户态的%eax的位置。然后，函数关中断，检查current->thread_info中的flags。

如果有flags置位，说明在返回用户态前，有别的工作要做。为了避免代码的重复，这种情况的处理是跳转到resume_userspace或work_pending labels处。本质上，iret汇编语言从内核栈中取得之前被sysenter_entry()保存的5个参数,把CPU切换到用户态和执行SYSENTER_RETURN label处的代码。

10.4. Parameter Passing

像通用函数那样，系统调用也需要一些输入输出参数，它们可能是确定的值，用户态进程的变量地址，甚至是包含了指向用户态函数的数据结构的地址。

因为system_call()和sysenter_entry()函数是Linux中所有系统调用的入口，它们至少有一个参数：系统调用号，通过%eax传递。

常规的C函数通常是通过把参数写入用户栈或内核栈来实现参数传递。而系统调用从用户态到内核态，没有用户栈和内核栈可以使用。因此，系统调用在调用之前把参数写入寄存器，然后内核把寄存器值拷贝进内核栈，接着调用系统调用服务程序，系统调用服务程序就是常规的C函数了。

然而，为了通过寄存器传递参数，2个条件必须满足：

1. 参数长度不能超过寄存器长度（32-bit）。
2. 参数个数不能超过6个，系统调用号存在%eax（不算在这6个参数中）。因为，X86只有有限的寄存器。

用来保存系统调用号和参数的寄存器有：%eax（存放系统调用号），%ebx（第一个参数），%ecx（第二个参数），%edx（第三个参数），%esi（第四个参数），%edi（第五个参数）和%ebp（第六个参数）。system_call()和sysenter_entry()是用

SACE_ALL宏把这些寄存器的值存放在内核栈中。所以，当系统调用服务程序在内核栈中，按照寄存器压栈顺序，就能依次读出系统调用的参数。内核栈的配置是按常规C函数调用时一样，因此系统调用服务程序能够容易的找到它的参数。

在少数情况下，即使系统调用不需要任何参数，相应的服务程序需要知道系统调用发出前的CPU寄存器的内容。例如，do_fork()函数需要知道系统调用发出前寄存器的值，这样才能复制它们给子进程。于是，一个pt_regs类型的参数允许服务程序访问保存在内核栈中的被SAVE_ALL保存的寄存器的值。

服务程序返回值保存在%eax中，这是被C编译器自动实现的。

10.4.1. Verifying the Parameters

所有的系统调用参数在被内核使用前必须被小心的检测。如果检测通不过，处理函数会返回一个负数。

对于系统调用一种比较通用的检测是，无论何时，一个参数指定了一个地址，内核必须检查它是否在进程地址空间。有两种方法实现这种检测：

1. 检测线性地址在进程地址空间中，以及包含它的memory region有合适的访问权限。
2. 检测线性地址比PAGE_OFFSET(0xC0000000)低(例如，它不能落在留给内核的地址中)。

第一种检测如果每次都进行，将很耗时，而且系统调用中参数错误是很少见的。因此。从Linux2.2开始，只进行第二种检测。这种方案高效的多。它只检测线性地址是否比PAGE_OFFSET(0xC0000000)小。至于线性地址是否在进程地址空间检测，被推迟到将线性地址转换为物理地址时，Page Fault异常可能会发生，异常处理函数会捕捉到系统调用中参数的线性地址的错误。

检测线性地址是否比PAGE_OFFSET(0xC0000000)小是至关重要的，因为内核地址空间可以访问所有的RAM，如果不进行这个检测，那么用户态进程就能通过系统调用访问所有物理页面而不引起Page Fault异常。

10.4.2. Accessing the Process Address Space

系统调用服务函数常需要读写进程地址空间的数据（用户态的数据）。Linux有一个宏集合来使这种操作更方便。我们将描述其中两种：get_user()和put_user()。第一个函数用来从用户空间读取1,2或4个连续字节，第二个则是写入1,2或4字节进用户空间。

每个函数接收2个参数，x(字节数),ptr(地址)。get_user(x, ptr)被扩展成__get_user_1(),__get_user_2()或__get_user_4()。
以__get_user_2()为例：

```
__get_user_2:
  addl $1, %eax
  jc bad_get_user
  movl $0xfffe000, %edx /* or 0xfffff000 for 4-KB stacks */
  andl %esp, %edx
  cmpl 24(%edx), %eax
  jae bad_get_user
  2:  movzwl -1(%eax), %edx
  xorl %eax, %eax
  ret
```

```
bad_get_user:
xorl %edx, %edx
movl $-EFAULT, %eax
ret
```

%eax寄存器包含了ptr地址，头6个指令执行类似与access_ok()宏的检测：他确保要读取的2bytes地址在4GB以下而且小于current->addr_limit.seq值。如果地址是合法的，函数执行movzwl指令把读取的数据存放在%edx的最低2字节，同时把%edx高2字节设为0，然后把0放在%eax返回。如果地址不合法，函数跳转到bad_get_user处。清空%edx，把%eax设为-EFAULT,结束。

10.4.3. Dynamic Address Checking: The Fix-up Code

当内核态发生Page Fault异常时，有4种情况可能出现，它们必须被异常处理函数区别对待：

1. 内核尝试寻址一个属于进程地址空间的页面，但相应的page frame要么不再内存中，要么是内核尝试写一个只读页面。在这种情况下，异常处理函数要分配一个新页面并适当的初始化。
2. 内核寻址的地址是它的地址空间，但相应的Page Table entry 没有被初始化（例如处理Noncontiguous Memory Area Accesses），这种情况下，内核必须恰当的设置当前进程的Page Tables entries。
3. 一些内核函数有编程错误，当程序执行时会产生Page Fault异常，或者是异常可能由短暂的硬件故障引起。当这种情况发生时，处理函数将引发一个内核oops。
4. 一个系统调用程序试图访问不属于进程地址空间的线性地址。

Page Fault处理函数可以很容易的识别出第一种情况:通过确认引发异常的线性地址是否在一个进程的memory region中。识别第二种情况:检测master kernel Page Table中对应线性地址的 entry是否为NULL。接下来将区别剩下两种情况。

10.4.4. The Exception Tables

内核只会用很窄一个范围内的能引发Page Fault异常的函数来访问进程地址空间。那么，如果异常是因为参数非法造成的，相应的引发异常的指令一定属于在一个小范围函数集合中的某函数。所以，不需很多代价就能把每一条内核访问进程地址空间的指令放入一个exception table结构。Page Fault异常处理就因此而方便了，如果一个异常发生在内核态，do_page_fault()处理函数检查exception table：如果包含触发异常的指令地址，那么错误是由非法的系统调用参数造成；否则，这是一个更严重的bug。

Linux定义了一些exception tables。他们由C编译器自动产生。保存在内核代码段的__ex_table部分，它的开始和结束地址由__start___ex_table和__stop___ex_table标记。

而且，每个内核动态加载的模块包含了它自己的local exception table。这个table由C编译器在创建模块镜像时由C编译器自动生成。

每个exception table的入口是个exception_table_entry结构，有两个成员：

1. insn:访问用户地址空间的指令的线性地址。
2. fixup：当Page Fault异常由在insn处的指令触发时，调用的汇编代码地址。

fixup代码由一些汇编指令组成，用来解决触发的异常引起的问题。将在后文看到，fix通常是插入一些指令序列，强迫服务程序返回一个错误码给用户态进程。这些指令，通常定义在访问进程地址空间的宏或函数处，被C编译成.fixup段。

search_exception_tables()函数用来在所有exception tables里搜索特定地址：如果地址在table里，函数返回指向exception_table_entry结构的指针；否则，返回NULL。Page Fault处理函数do_page_fault()执行下列代码：

```
if ((fixup = search_exception_tables(regs->eip))) {
    regs->eip = fixup->fixup;
    return 1;
}
```

regs->eip存放的是异常在内核态发生时%eip的值。如果它在exception table中，do_page_fault()用fix代码地址代替当前%eip中的值。然后Page Fault处理函数退出，被中断的程序恢复执行,不过是从fixup代码开始执行。

Chapter 11. Signals

11.1. The Role of Signals

一个signal是一个很短的信息，可以发送给一个进程或一组进程。传递给进程的唯一信息是一个代表signal的数字；在标准的signal中没有额外的空间保存其它附加信息。

Signal主要为两个目的服务：

1. 当特定事件发生时通知进程。
2. 促使进程执行一个signal处理函数。

Linux2.6在X86体系下的regular signal有31个。POSIX标准还引进了一类新的signals,被称作real-time signals；它们的signal数字在Linux下是从32到64。它们之所以与regular signal不同是在于，当多个同种sugnals被接受时，它们总是组织成队列。换句话说，同种regular signals是不会组织成队列的，那么如果regular signal被发送多次，只有一次会被接收进程接受。虽然Linux不使用real-time signals，但它通过一些特殊的系统调用，完整的支持了POSIX标准。

Signal的一个重要的特性是，它们可以在任意时刻发送给进程，进程当时的状态是不可预计的。发送给进程的signals不是立即被处理，而是被内核保存下来，直到进程恢复运行。

内核把signal传输分成了两阶段：

1. Signal generation:内核更新目的进程的数据结构，代表一个新的signal被发送。
2. Signal delivery:内核通过改变执行状态，开始执行特殊的signal处理函数，强迫目标进程对signal做出反应。

每个产生的signal能被最多delivered一次。Signals是可被消耗的资源：一旦它们被delevered, all process descriptor information that refers to their previous existence is canceled.

已经被产生的signals但还未被delivered称做pending signals。在任何时刻一个进程上，给定类型的signal最多只有一个pending signal存在；发给同进程同类型的更多的signals被简单的丢弃。但real-time signals不同：他们能有同种类型的多个pending signals。

通常的，一个signal可能pending一个不可预计的时间。如下因素应被考虑到：

1. signals通常被仅仅被让前运行进程来delivered。
2. 给定类型的signal可以被进程选择性的阻塞掉。在这种情况下，进程在移除阻塞才能接收这种signal。
3. 当一个进程执行signal处理函数时，它通常屏蔽掉同种signal。一个signal处理函数不能被另一个同时发生的同种信号处理中断，因此处理函数不需要被重入。

一个pending signal如果是发给特定进程的，那它是私有的，如果是发送给整个线程组，则是共享的。

11.1.2. POSIX Signals and Multithreaded Applications

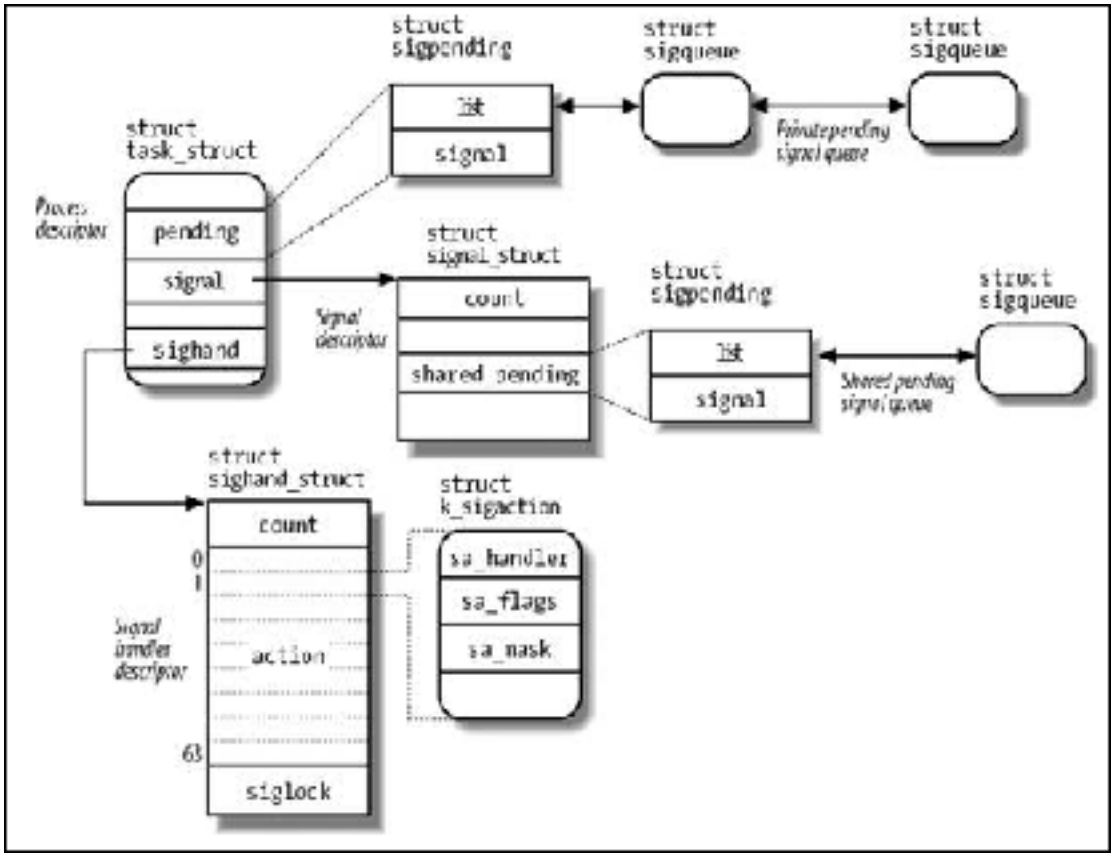
POSIX 1003.1标准对多线程应用的signal处理有一些严格的要求：

1. signals处理函数必须被所有多线程应用中各个线程共用；然而，每个线程必须有自己的mask of pending和blocked signals。
2. kill()和sigqueue()函数必须发送信号给整个多线程应用，而不是特定线程。同样的规则适用于内核发出的signal。
3. 每个发送到多线程应用的signal必须只能被一个线程delivered，这个线程是被内核随意选择的没有阻塞该signal的线程。
4. 当一个fatal signal发送给多线程应用，内核将杀死所有线程。

11.1.3. Data Structures Associated with Signals

对系统中每个进程，内核必须追踪什么signals正在pending或masked；内核必须追踪每个线程组是如何处理每个signal的。为了实现这个目的，内核使用了多种数据结构存放在进程描述符中。进程描述符中于signal相关的的成员有：

1. signal：指向进程的signal descriptor。
2. sighand:指向进程的signal处理descriptor。
3. blocked:mask of blocked signal。
4. pending:存放私有pending signal的数据结构。
5. sas_ss_sp:可选的signal handler stack地址。
6. sas_ss_size：可选的signal handler stack大小。
7. notifier:指向被设备驱动使用的函数来阻塞进程的某些signals。
8. notifier_data:指向被notifier函数使用的数据。
9. notifier_mask:被设备驱动用notifier阻塞的bit mask of signals。



blocked成员主要数据结构是sigset_t:

```
typedef struct {
  unsigned long sig[2];
} sigset_t;
```

Linux支持的signals最多64个。1-31保存在sig[0]中，是regular signal。Sig[1]中是real-time signal。

11.1.3.1. The signal descriptor and the signal handler descriptor

进程描述符中的signal指向signal descriptor，追踪了共享的pendng signals。事实上，signal descriptor还包含了rlim，代表了每个进程的资源上限数组，以及pgrp和session域，分别代表了组leader的PID和进程的session leader的PID。signal descriptor被所有属于同一个线程组的所有进程共享。

signal descriptor的主要数据结构是signal_struct,与signal相关的主要成员是:

1. count: signal descriptor计数器。
2. live:线程组中活着的进程数量。
3. wait_chldexit:通过wait4()系统调用进入睡眠的进程的等待队列。
4. curr_target:线程组中最后一个收到signal的进程描述符。
5. shared_pending:保存共享pending signals的数据结构。
6. group_exit_code:对线程组的进程结束码。
7. group_exit_task:当杀死整个线程组时使用。
8. notify_count:当杀死整个线程组时使用。
9. flags: 当正在delivering 的signals修改了进程状态时，使用。

每个进程还有一个signal handler descriptor,描述了每个signal如何被线程组处理。主要数据结构sighand_struct,主要成员是:

1. count: signal handler decriptor使用计数器。
2. action:64个元素的数组，指向了在delivering signals时的动作。
3. siglock:spin lock 保护sigenal descriptor和signal handler descriptor。

在POSIX标准中，所有线程组中的轻量级进程指向同一个signal descriptor和相同的signal handler descriptor。

11.1.3.2. The sigaction data structure

X86平台，所有的signal属性对用户态进程都是可见的。因此，k_sigaction结构中只有一个sigaction类型的sa变量,它包含了如下成员:

[*]被用户态应用用来传递参数给signal()和sigaction()系统调用的sigaction结构，稍稍有些不同与被内核使用的sigaction，虽然它们包含了基本相同的信息。

1. sa_handler:指定了动作类型。它的值能指向SIG_DFL(值为0)指示使用缺省函数，或SIG_IGN(值为1)指示了signal被忽略。
2. sa_flags:flags集合，指示了signal如何被处理。
3. sa_mask:指示了被屏蔽的signal。

11.1.3.3. The pending signal queues

为了追踪哪些signal当前在pending中，内核为每个进程维护了2个pending signal队列。

1. shared pending signal queue: signal descriptor的shared_pending成员，存放着整个线程组的pending signals。
2. private pending signal queue: 进程描述符的pending成员，存放着特定进程的pending signals。

一个pending signal queue由一些sigpending数据结构组成，其定义为:

```
struct sigpending {
   struct list_head list;
   sigset_t signal;
}
```

signal成员是一个bit mask，指定了pending signals，结构通过list链入sigqueue数据结构。sigqueue主要成员有:

1. list:pengding signal queue队列头。
2. flags: sigqueue的flags。
3. info: 描述产生signal的事件。
4. user: 指向进程拥有者的数据结构。

siginfo_t数据结构长128字节，存储了关于特定signal发生的信息；它包括如下成员:

1. si_signo: signal号。
2. si_errno: 导致signal产生的指令的出错码，0代表没有错误。
3. si_code: 一个code代表了谁产生了signal。
4. _sifields: 一个union，存放的信息依赖与signal类型。列如，siginfo_t数据结构代表的是SIGKILL signal,那么存放在_sifields的是发送进程的PID和

UID。相反，siginfo_t代表的是SIGSEGV signal发生，存放在_sifields的是导致signal发生的内存地址。

11.2. Generating a Signal

11.2.1. The specific_send_sig_info( ) Function

specific_send_sig_info()函数发送一个signal给特定进程。它有3个参数：

1. sig:signal号。
2. info:要么是siginfo_t表的地址，要么是3个特殊值：0代表signal从用户进程发送。1代表被内核发送。2代表被内核发送，signal是SIGSTOP或SIGKILL。
3. t：指向目标进程的描述符。

specific_send_sig_info()函数要求关闭中断，得到t->sighand->siglock spin lock。函数执行下列步骤：

1. 检测进程是否忽略signal；如果是，返回0（不产生信号）。当下列3个条件都满足时，signal被忽略：
   - 进程没有正在被traced(t->ptrace的PT_PTRACE清0)。
   - signal没有被阻塞（sigismember(&t->blocked, sig)返回0）。
   - signal要么被显式忽略(t->sighand->action[sig-1]->sa_handler为SIG_IGN)或隐式的忽略(t->sighand->action[sig-1]->sa_handler为SIG_DFL并且signal是SIGCONT,SIGCHLD,SIGWINCH,或SIGURG)。
2. 检查signal是 non-real-time(sig<32)并且同类型的sigal已经pending在进程的private pending signal queue（sigismember(&t->pending.signal,sig) 返回1）中：如果是，函数不做任何事，只是返回0。
3. 调用send_signal(sig, info, t, &t->pending)把signal加到进程的pending signals集合；
4. 如果send_signal()成功结束，并且signal没有被阻塞（sigismember(&t->blocked,sig)返回0），调用signal_wake_up()函数通知进程pending signal的到来。signal_wake_up()执行下列步骤：
   - 设置t->thread_ibfo->flags为TIF_SIFPENDING flags。
   - 调用try_to_wake_up()，如果进程是在TASK_INTERRUPTIBLE或TASK_STOPED并且signal是SIGKILL，则进程被唤醒。
   - 如果try_to_wake_up()返回0，说明进程是runnable：如果是，检查进程是否在另一个CPU上执行，如果是，发送一个interprocessor interrupt中断给那个CPU，强迫其进行一次调度。因为进程在schedule()返回时会检查是否有signal pending,因此interprocessor interrupt一定会导致目标进程很快的注意到这个pending signal。
5. 返回1（信号成功产生）。

11.2.2. The send_signal( ) Function

send_signal()函数插入一个新项进入pending signal queue。它接收参数：signal号sig，siginfo_t结构地址info（或特殊值：0代表signal从用户进程发送。1代表被内核发送。2代表被内核发送，signal是SIGSTOP或SIGKILL），目标进程的进程描述符t和要加入的pending signal queue的地址signals。

函数执行下列步骤：

1. 如果info值是2，说明signal类型要么是SIGKILL要么是SIGSTOP，而且是被内核通过forc_sig_specific()产生：在这种情况下，它跳转到步骤9。这种情况需要内核立即作出反应，而不需要添加到pending signal queue中。
2. 如果目标进程的用户的pending signal数目(t->user->sigpending)小于当前进程的资源上限(t->signal->rlim[RLIMIT_SIGPENDING].rlim_cur)，函数分配一个sigqueue数据结构给新signal：
   ```
   q = kmem_cache_alloc(sigqueue_cachep, GFP_ATOMIC);
   ```
3. 如果目标进程的用户的pending signal数目太多或分配内存失败，跳转到步骤9。
4. 递增目标进程的用户的pending signal数目(t->user->sigpending)和递增t->user的引用计数。
5. 把sigqueue数据结构加入到pending signal queue中：
   ```
   list_add_tail(&q->list, &signals->list);
   ```
6. 填充sigqueue数据结构中的siginfo_t。
   ```
   if ((unsigned long)info == 0) {
       q->info.si_signo = sig;
       q->info.si_errno = 0;
       q->info.si_code = SI_USER;
       q->info._sifields._kill._pid = current->pid;
       q->info._sifields._kill._uid = current->uid;
   } else if ((unsigned long)info == 1) {
       q->info.si_signo = sig;
       q->info.si_errno = 0;
       q->info.si_code = SI_KERNEL;
       q->info._sifields._kill._pid = 0;
       q->info._sifields._kill._uid = 0;
   } else
           copy_siginfo(&q->info, info);
   ```
   copy_siginfo () 函数拷贝参数中siginfo_t结构的内容。
7. 在bit mask of queue中设置相应的signal。
8. 返回0：signal成功的加入到pending signal queue。
9. 到这一步骤，signal没有加入signal pending queue，这是因为pending signals太多了或者没有足够的内存分配给sigqueue数据结构，或者signal需要被内核立即强制执行。如果signal是real-time并且是被明确要求加入队列的内核函数发送，函数返回错误码：-EAGAIN。
   ```
   if (sig>=32 && info && (unsigned long) info != 1 &&
       info->si_code != SI_USER)
       return -EAGAIN;
   ```
10. 设置bit mask of queue的对应signal号：
    ```
    sigaddset(&signals->signal, sig);
    ```
11. 返回0：即使signal没有加入队列，相应的bit已经被设置。

即使没有足够的内存来分配sigqueue结构，让目标进程接收到signal是很重要的。例如：假设进程消耗了太多的内存，kill()系统调用应成功发送，即使没有足够的内存分配sigqueue结构。否则，系统管理者将无法删除该进程，而释放内存。

11.2.3. The group_send_sig_info( ) Function

group_send_sig_info()函数发送一个signal给整个线程组。它有3个参数：signal号sig，siginfo_t结构（及3个特殊值）info，进程描述符p。

函数执行下列步骤：

1. 检查sig是否合法：
   ```
   if (sig < 0 || sig > 64)
   return -EINVAL;
   ```

2. 如果signal是被用户态进程发送的，那么检查这个操作是否允许。signal至少满足以下一个条件才能被delivered。
   - 发送进程的用户有合适的权限。
   - Signal是SIGCONT并且目标进程和发送进程在同一个login session。
   - 进程属于同一用户。

如果用户态进程不允许发送signal，函数返回-EPERM。

3. 如果sig参数等于0,函数立即返回，不产生任何signal。
4. 请求p->sighand->siglock spin lock和关中断。
5. 调用handle_stop_signal()函数，它检测出一些signal会使在目标线程组的pending signal失效。
   a. 如果线程组被杀死了（signal descriptor的flags置为SIGNAL_GROUP_EXIT），函数返回。
   b. 如果sig是SIGSTOP, SIGTSTP, SIGTTIN, 或 SIGTTOU signal，函数调用rm_from_queue()从shared pending signal queue中和线程组中所有成员的private queues中移除SIGCONT signal。
   c. 如果signal是SIGCONT，调用rm_from_queue()从shared pending signal queue中和线程组中所有成员的private queues中移除SIGSTOP, SIGTSTP, SIGTTIN, 或 SIGTTOU signal，并且唤醒这些进程。

```
rm_from_queue(0x003c0000, &p->signal->shared_pending);
t = p;
do {
    rm_from_queue(0x003c0000, &t->pending);
     try_to_wake_up(t, TASK_STOPPED, 0);
     t = next_thread(t);
     } while (t != p);
```

6. 检查线程组是否要忽略signal。如果是，返回0。当满足所有3个条件时，signal被忽略。
7. 检查signal是否是non-real-time并且同样的signal已经pending在线程组的shared pending queue：如果是，直接返回0。
8. 调用send_signal()把signal添加到shared pending queue。
9. 调用__group_complete_signal()唤醒线程组中的一个轻量级线程。
10. 释放p->sighand->siglock spin lock并且开中断。
11. 返回0。

__group_complete_signal()函数扫描线程组中的进程，寻找一个进程接收新signal。一个进程如果满足如下所有条件可能被选取。

1. 该进程没有阻塞signal。
2. 进程不在EXIT_ZOMBIE, EXIT_DEAD, TASK_TRACED, or TASK_STOPPED状态（有个例外：signal是SIGKILL，进程可以是TASK_TRACED, or TASK_STOPPED）。
3. 进程没有正在被kill。即，PF_EXITING flag没有被设置。
4. 要么进程在CPU上执行，要么TIF_SIGPENDING没有被置位。

一个线程组中可能有多个进程满足上面条件，函数按如下原则选择其中之一：

1. 如果作为参数传给group_send_sig_info满足条件，选择它。
2. 否则，扫描线程组成员，从p->signal->curr_target开始扫描，选择第一个。

如果__group_complete_signal()成功的选择到了一个合适进程。函数首先检查signal是否是fatal：如果是，通过给每个线程发送SIGKILL将整个线程组杀死，否则函数调用signal_wake_uo()来通知选择的进程，有新pending signal到来。

11.3. Delivering a Signal

内核在完成中断处理和异常处理后，返回用户态时会检测是否TIF_SIGPENDING置位。为了处理未屏蔽的pending signals，内核调用do_signal()函数，它接收2个参数：
regs:略。
oldset:函数用来保存bit mask array of blocked signals的变量地址，如果它是NULL，则不需要保存bit mask array。

do_signal()函数通常仅在CPU即将返回用户态时被调用。因此，如果一个中断处理函数调用do_signal()，do_signal()函数简单的返回。

```
if ((regs->xcs & 3) != 3)
   return 1;
```

如果oldset参数为NULL，函数用current->blocked地址初始化它。

```
if (!oldset)
   oldset = &current->blocked;
```

do_signal()函数中，调用get_signal_to_deliver()来取得一个pending signal。它的核心由一个循环组成，循环不停的重复调用dequeue_signal()函数，直到没有nonblocked pending signals在私有和共享pending signal queues中，或者找到了一个合适的signal。dequeue_signal()的返回值保存在signr局部变量中，再由get_signal_to_deliver()返回给do_signal()。如果它的值为0，意味着所有的pending signals都被处理完了，do_signal()能够结束了。如果dequeue_signal()返回了一个非0值，说明还有pending_signal在等待处理，do_signal()会处理该pending signal。

让我们来看看do_signal()如何处理每个pending signal。首先，它检查当前接收进程是否被其它进程监视着；这种情况下，do_signal()调用do_notify_parent_cldstop()和schedule()来使监视进程注意到signal处理。

然后，do_signal()执行：ka = &current->sig->action[signr-1];
依靠ka中的内容，3种可能的动作之一将被执行：忽略signal，执行一个缺省的动作，或者执行一个signal处理函数。

11.3.1. Executing the Default Action for the Signal

如果ka->sa.sa_handler等于SIG_DFL，do_signal()必须执行缺省的动作。唯一的例外是如果接收进程是init(1号进程)，任何signal不将被处理：

```
if (current->pid == 1)
    continue;
```

对其他进程，如果signal缺省动作是"ignore"，简单的进行处理：

```
if (signr==SIGCONT || signr==SIGCHLD ||
    signr==SIGWINCH || signr==SIGURG)
continue;
```

缺省动作是"stop"，将停止线程组中所有进程。为了达到这个目的，do_signal()把进程状态设置为TASK_STOPPED，然后调用schedule()：

```
if (signr==SIGSTOP || signr==SIGTSTP ||
        signr==SIGTTIN || signr==SIGTTOU) {
    if (signr != SIGSTOP &&
```

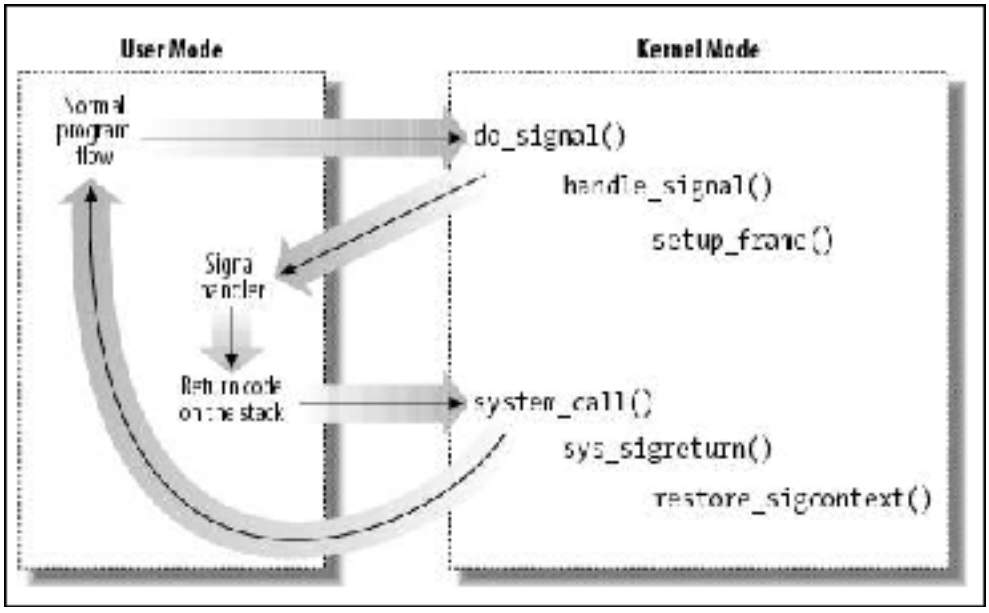
```
        is_orphaned_pgrp(current->signal->pgrp))
        continue;
    do_signal_stop(signr);
}
```

11.3.2. Catching the Signal

如果signal处理函数被用户进程设置了,do_signal()函数必须强制执行它。调用handle_signal():

```
handle_signal(signr, &info, &ka, oldset, regs);
if (ka->sa.sa_flags & SA_ONESHOT)
    ka->sa.sa_handler = SIG_DFL;
return 1;
```

如果SA_ONESHOT置位，重设该signal处理函数为SIG_DFL。那么下次该signal发生时，就无法再次用之前的处理函数处理。注意到，do_signal()处理完该signal后将返回。其它pending signal不会被处理，直到do_signal()再次被调用。这个方法保证了real-time signals被按照合适的顺序处理。



11.3.2.1. Setting up the frame

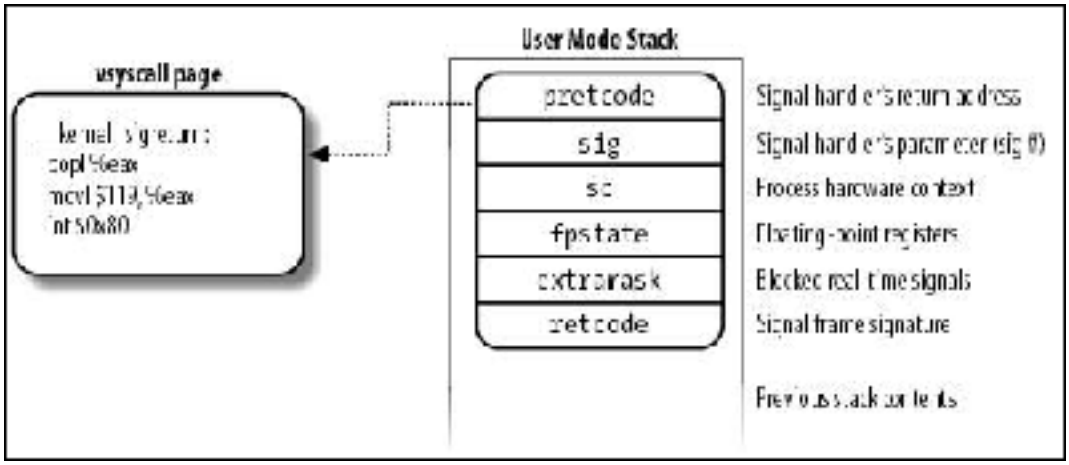
为了正确的设置进程的用户栈，handle_signal()函数调用setup_frame()(使用在不需要siginfo_t表的signal处理)或setup_rt_frame()(使用在需要siginfo_t表的signal处理)。内核通过检查signal的sigaction->sa_flags的SA_SIGINFO标记位来判断使用哪个函数。

setup_frame()函数接收4个参数:

1. sig:signal号.
2. ka:与signal相关的k_sigaction表的地址。
3. oldest: Address of a bit mask array of blocked signals
4. regs:用户态寄存器保存在内核栈中的地址。

setup_frame()函数压入用户栈的数据结构叫做一个frame，它包含了处理signal和正确返回到sys_sigreturn()的必要信息。一个frame是个sigframe表，包含了下列域:

pretcode:signal处理函数的返回地址；它指向__kernel_sigreturn处的代码。
sig:signal号；是signal处理函数的参数。
sc:sigcontext数据结构，包含了用户态进程切换进内核态前硬件上下文(从crrent内核栈中拷贝出来的信息)。它也包含了a bit array指定了进程的被屏蔽的regular signals。
fpstate:_fpstate数据结构，用来存放用户态进程的浮点寄存器。
extramask:bit array指定了被阻塞的real-time signals。
retcode:8字节代码，发出sigreturn()系统调用。在Linux2.6中，它只被用来作为signature,从而debuggers能识别出signal stack frame。



setup_frame()函数先调用get_sigframe()计算出frame的起始地址。Frame是存放在用户栈中，因此函数返回如下值:
(regs->esp - sizeof(struct sigframe)) & 0xfffffff8
因为用户栈是向下生长的，所以初始地址是栈顶减去frame大小，然后再对齐成8的倍数。

get_sigframe()返回的地址由accesee_ok宏来检测，如果合法，函数反复调用__put_user()填充frame的所有域。frame的pretcode初始化为&__kernel_sigreturn，这个地址指向的代码存放在vsyscall page中。

以上工作完成后，函数修改内核栈的regs区域，确保当cuurent在用户态恢复运行时能执行signal处理函数。

```
regs->esp = (unsigned long) frame;
regs->eip = (unsigned long) ka->sa.sa_handler;
regs->eax = (unsigned long) sig;
regs->edx = regs->ecx = 0;
regs->xds = regs->xes = regs->xss = _ _USER_DS;
regs->xcs = __USER_CS;
```

setup_frame()函数重新设置了保存在内核栈里的段寄存器。现在，signal处理函数需要的信息都在用户栈顶了。

setup_rt_frame()函数与setup_frame()类似，但是它在用户栈放入了一个extended frame(保存在rt_sigframe数据结构中)，extended frame包含了与signal相联系的siginfo_t表的内容。而且，函数把pretcode设为指向vsyscall page中的__kernel_rt_sigreturn代码。

11.3.2.2. Evaluating the signal flags

在设置好用户栈后，handle_signal()函数检测signal的flags，如果SA_NODEFER flag没有设置，在sigaction->sa_mask中的signal必须在执行signal处理函数时被屏蔽。

```
if (!(ka->sa.sa_flags & SA_NODEFER)) {
    spin_lock_irq(&current->sighand->siglock);
    sigorsets(&current->blocked, &current->blocked, &ka-
>sa.sa_mask);
    sigaddset(&current->blocked, sig);
    recalc_sigpending(current);
    spin_unlock_irq(&current->sighand->siglock);
    }
```

sigorsets()函数从用户的信号屏蔽码刷新阻塞信号集，可见信号有两个地方屏蔽,第一是进程的task_struct中有一个block,这是一直起作用的.还有就是设置信号处理程序时的sigaction结构的sa_mask.kill,stop信号不允许被屏蔽.当然也不能设置处理函数。

recalc_sigpending()函数检查进程是否有非屏蔽的pending signals,如果有，则设置TIF_SIGPENDING标志。

11.3.2.3. Starting the signal handler

当do_signal()返回时，当前进程恢复在用户态执行。因为之前被setup_frame()的准备工作，现在%eip指向signal处理函数的第一条指令，%esp指向用户栈中frame的起始内存地址。结果是，signal处理函数开始执行了。

11.3.2.4. Terminating the signal handler

当一个signal处理函数完成时，返回地址由frame中的precode指向vsyscall page:

```
__kernel_sigreturn:
  popl %eax
  movl $__NR_sigreturn, %eax
  int $0x80
```

所以，signal号被从栈中丢弃了；sigreturn()系统调用被执行。

sigreturn()函数计算pt_regs数据结构regs的地址，它包含了用户态进程的硬件上下文。从frame中计算:

```
frame = (struct sigframe *)(regs.esp - 8);
if (verify_area(VERIFY_READ, frame, sizeof(*frame)) {
    force_sig(SIGSEGV, current);
    return 0;
}
```

接着，函数把bit arrar of signal从frame的sc域拷贝到current->blocked。结果是，所有被signal处理函数执行时屏蔽的signals are unblocked。recalc_sigpending()接着调用。

sys_sigreturn()函数必须把进程上下文从frame的sc域拷贝到内核栈，然后从用户栈中移除frame;这两个任务是用rsstore_sigcontext()函数完成的。

rt_sigqueueinfo()类似。

11.3.3. Reexecution of System Calls

系统调用请求的资源不是总是能被内核满足；当不能满足时，内核把进程转入TASK_INTERRUPTIBLE 或TASK_UNINTERRUPTIBLE状态。

如果进程在TASK_INTERRUPTIBLE状态并且有其它进程发送signal给它。内核把进程转换成TASK_RUNNING状态而并不让它完成系统调用,进程会直接返回,再返回用户空间前会进行signal处理，然后返回EINTR, ERESTARTNOHAND, ERESTART_RESTARTBLOCK, ERESTARTSYS, 或ERESTARTNOINTR错误码.

实际中，用户态进程能接收到的唯一出错码是EINTR，代表着系统调用没有完成。其它几种错误码被内核使用，来指定是否在signal处理完成后自动重新执行系统调用。

当我们delivering一个signal，内核必须确定进程真的是之前发出了一个系统调用。这是通过regs的orig_eax域判断：
1. 中断：orig_eax包含的是中断号减去256。
2. 0x80异常(也包括sysenter)：orig_eax包含的是系统调用号。
3. 其它异常：orig_eax包含的是-1。

所以，一个非负的orig_eax值意味着signal在系统调用中睡眠，状态是TASK_INTERRUPTIBLE。

通过如下指令可以使系统调用重新执行：
```
regs->eax = regs->orig_eax;
regs->eip -= 2;
```

Chapter 12. The Virtual Filesystem

12.1. The Role of the Virtual Filesystem (VFS)

虚拟文件系统是一个内核软件层，它处理了所有与标准Unix文件系统相关的系统调用。它的主要作用是提供了一个通用的接口给许多种文件系统。

commom file model由下列对象组成：
1. The superblock object:保存了关于mounted文件系统的信息。对于基于磁盘的文件系统，这个对象等同于一个存储在磁盘上的filesystem control block。
2. The inode object:保存了关于某文件的通用信息。对于基于磁盘的文件系统，这个对象通常等同于一个磁盘上的file control block。每个inode对象与一个inode号关联，它唯一的代表了文件系统中的一个文件。
3. The file object:保存了关于打开文件和进程之间联系的信息。这个信息仅仅当进程拥有打开文件时，存在于内核内存中。
4. The dentry object:保存了关于目录和对应文件之间链接的信息。每个基于磁盘的文件系统在磁盘上用各自不同的方式保存了该信息。

12.2. VFS Data Structures

12.2.1. Superblock Objects

一个超级块对象由super_block结构组成。所有的超级块对象被链入一个循环链表，链表头：struct list_head super_blocks。超级块通过super_block->s_list挂入该链表。

super_block->s_fs_info指向一个属于特定文件系统的超级块信息;例如，如果超级块对象引用的是Ext2文件系统，s_fs_info指向的就是ext2_sb_info结构，它包含了磁盘分配bit masks和其它与VFS common file model相关的数据。

通常的，s_fs_info指向的内容是对磁盘信息的在内存中的一个副本,这是由于效率的原因。这导致了内存中超级块和磁盘中对应信息的不同步。super_block->s_dirt flag指示了超级块是否dirty。为了防止用户突然断电时对文件系统的损害，Linux周期性的把所有dirty超级块拷贝进磁盘。

与超级块关联的操作叫superblock operations。由super_operations数据结构表示，superblock operations包含在super_block->s_op中。

12.2.2. Inode Objects

磁盘上的inode数据结构包含了文件系统处理一个文件所需要的所有信息。一个文件名是一个临时分配的标志，能够被更改，但inode唯一的代表了一个文件，只要文件还存在，inode就不会改变。inode对象在内存中由struct inode组成。

每个内存中的inode对象是磁盘上对应inode的副本。当inode->i_state等于I_DIRTY_SYNC, I_DIRTY_DATASYNC, 或 I_DIRTY_PAGES,说明inode是dirty,相应的磁盘上的inode必须更新。I_DIRTY宏能检测这三个值。inode->i_state其它可能的值有:I_LOCK(inode正在被I/O传输使用),I_FREEING(inode对象被释放),I_CLEAR(inode内容不再有意义),I_NEW(indoe对象已经被分配，但是还没有从磁盘inode中读取数据来填充)。

每个inode对象总是出现在下列队列之一中(都是通过inode->i_list链入其中一个队列):

1. 有效的但未使用inodes链表。这些inodes非dirty并且i_count为0。链表头保存在inode_unused变量中。这个队列表现为一个disk cache。
2. 正在使用inodes链表。这些链表非dirty并且i_count是正数。连表头放在inode_in_use变量中。
3. dirty inodes链表。存放在该inode对应的超级块的s_dirty链表中。

每个inode对象还通过inode->i_sb_list挂入了对应超级块的s_inodes链表中。

最后,inode对象还被包括在一个hash表中，名为inode_hashtable。当内核知道inode号和包含该文件的文件系统的超级块的地址时,hash表加速了inode对象的搜索速度。为了解决hash表的碰撞问题，通过inode->i_hash挂入链表。

与一个inode对象相关联的操作叫inode operations。它们被inode_operations数据结构描述。由inode->i_op指向该数据结构对象。

12.2.3. File Objects

file对象描述了一个进程与它打开的文件之间的联系。file对象在文件打开时被创建，由file数据结构构成。file对象在磁盘中没有相应镜像，因此不需要"dirty"成员。

file对象中的主要信息是file pointer。它代表了对磁盘文件的操作的下一个位置。因为可能会有许多不同进程同时操作文件，因此file pointer必须保存在每个file对象中而不是保存在inode对象中。

file对象通过名为filp的slab cache分配,这个cache的描述符保存在filp_cachep变量中。files_stat变量的max_files成员指定了可分配的file对象的最大数量,也是指系统中在同一时间能被同时访问的file数量。

每个"IN use"的file对象，通过file->f_file链入文件所在文件系统的超级块的s_files链表中。因此，属于不同文件系统的file对象被链接在不同队列中。files_lock spin lock保护了超级块的s_files链表，防止多个CPU同时访问它。

当VFS必须代替进程打开一个文件时,它调用get_empty_filp()函数分配一个新的file对象。函数调用kmem_cache_alloc()从filp cache中得到一个空闲的file对象，然后初始化它。

每个文件系统有自己的一个file operations集合，它执行了例如读文件和写文件之类的操作。当内核把一个inode从磁盘装载进内存，内存中的inode->i_fop指向了file operations, file operations的数据结构是file_operations。当一个进程打开文件时，VFS用inode->i_fop来初始化file->f_op。这样，下一步文件操作就能使用这些操作了。如果需要，VFS可以在以后通过重置f_op来修改file operations集合。

12.2.4. dentry Objects

VFS把目录看成了一个文件，这个文件包含了其它目录和文件的链表。一旦一个directory entry被读入内存，就被VFS转换成一个dentry对象，数据结构是dentry。内核为路径名的每个部分创建一个dentry对象。

dentry对象在磁盘上没有对应镜像。Dentry对象存放在名为dentry_cache的slab cache中。

每个dentry对象处于4种状态之一:

1. Free:dentry对象包含无效数据并且不再被VFS使用。对应的内存被slab allocator处理。
2. Unused:dentry当前没有被内核使用。对象的d_count使用计数器为0,但d_inode仍然指向相应的inode。Dentry对象包含了合法数据，但是如果为了回收内存,它的内容可能被丢弃。
3. In use:dentry对象当前被内核使用着。d_count使用计数是正数,d_inode指向相应的inode对象。dentry对象包含了有效信息并且不能被丢弃。
4. Negative:与dentry相关的inode不存在，要么是因为对应磁盘上的inode被删除了，要么是because the dentry object was created by resolving a pathname of a nonexistent file。dentry->d_inode被设为NULL,但是这个对象仍然留在dentry cache中，以后查询同一个路径名就能很快建立相应dentry。

与dentry相关的操作叫dentry operations,数据结构为dentry_operations，由dentry->d_op指向。

12.2.5. The dentry Cache

为了最有效的处理dentries,Linux使用了一个dentry cache,它由两种数据结构组成:

1. 处于in_use,unused,或negative状态的dentry对象集合。

2． 一个hash表，从一个给定的文件名和一个给定的目录可以快速的找到对应dentry。

dentry cache也表现为对一个inode cache的控制器。内核内存中的与unused dentries 相联系的inodes不会被丢弃，因为dentry cache仍然在使用它。那么，inode对象保存在RAM中能被对应dentries直接找到。

所有的"unused" dentries被包含在"Least Recently Used"链表中。换句话说，最近被释放的dentry对象被放在了链表的前面，因此最近最少实用的dentry对象被挂在链表最后。当dentry cache必须收缩时，内核从链表尾移除元素。dentry对象通过d_lru链入dentry_unused变量。

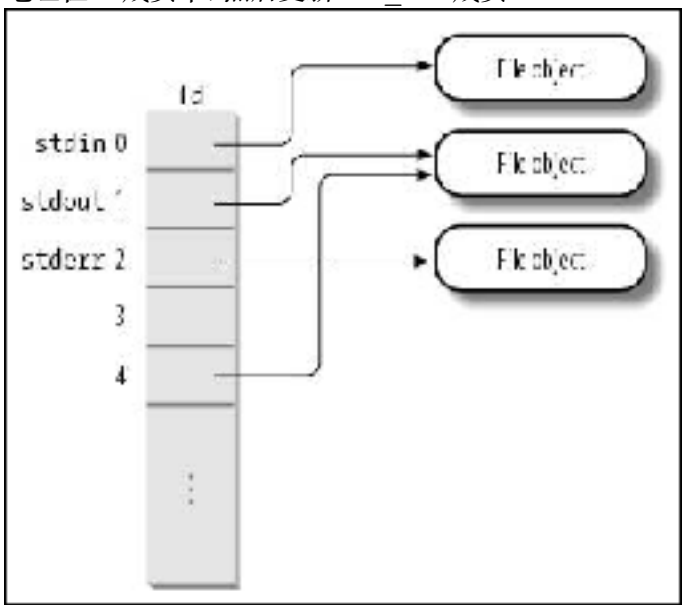
每个"in use" dentry 对象通过d_alias被链入对应inode对象的i_dentry域(因为一个inode可能与多个硬链接相连)。

一个"in use" dentry，当最后一个对应文件的硬链接被删除时，可能变为"negative"状态。这种情况下，dentry对象被移动到unused dentries LRU链表末端。每当内核收缩dentry cache时，移动到LRU链末尾的dentries逐渐的被释放。

12.2.6. Files Associated with a Process

数据结构fs_struct维护了进程和文件系统之间的关系。

数据结构files_struct维护了进程和打开文件之间的关系。
files_struct->fd指向了一个指向file对象的指针数组。数组大小保存在files_struct->max_fds。通常，fd指向files_struct->fd_array，这个数组包含了32个指针。如果进程打开超过32个文件，内核分配一个新的更大的指针数组并且保存其地址在fd成员中；然后更新max_fds成员。



12.3. Filesystem Types

12.3.1. Special Filesystems

Table 12-8. Most common special filesystems		
Name	**Mount point**	**Description**
bdev	none	Block devices (see Chapter 13)
binfmt_misc	any	Miscellaneous executable formats (see Chapter 20)
devpts	*/dev/pts*	Pseudoterminal support (Open Group's Unix98 standard)
eventpollfs	none	Used by the efficient event polling mechanism
futexfs	none	Used by the futex (Fast Userspace Locking) mechanism
pipefs	none	Pipes (see Chapter 19)
proc	*/proc*	General access point to kernel data structures
rootfs	none	Provides an empty root directory for the bootstrap phase
shm	none	IPC-shared memory regions (see Chapter 19)
mqueue	any	Used to implement POSIX message queues (see Chapter 19)
sockfs	none	Sockets
sysfs	*/sys*	General access point to system data (see Chapter 13)
tmpfs	any	Temporary files (kept in RAM unless swapped)
usbfs	*/proc/bus/usb*	USB devices

特殊文件系统不是与物理块设备绑定的。内核给每个安装的特殊文件系统分配一个假想的块设备，其major设备号为0，minor设备号任意(与其它特殊文件系统的minor设备号不同)。

12.3.2. Filesystem Type Registration

每个注册了的文件系统，其数据结构为file_system_type。

所有的文件系统对象通过file_system_type->next一个插入单链表。链表头保存在file_system变量中。File_system_lock 读/写 spin lock保护着这个链表。

file_system_type->fs_supers代表着一个属于该文件系统的超级块对象链表。超级块对象(比如属于同种文件系统的不同设备)通过s_instances挂入这个链表。

file_system_type->get_sb指向一个filesystem-type-dependent函数，它可以一个分配新的超级块对象并初始化它。

file_system_type->fs_flags储存了一些flags。

12.4. Filesystem Handling

每个文件系统有自己的root目录。The directory on which a filesystem is mounted is called the mount point.安装的文件系统是安装点目录所在文件系统的孩子。安装的文件系统的root目录隐藏了父文件系统安装点目录里的内容。

12.4.1. Namespaces

Linux2.6中，每个进程可以有自己的安装的文件系统树,这被称做进程的namespace。

通常大多数进程共享同样的namespace，这颗树的根是系统的root文件系统，被init进程使用。然而，一个进程如果被clone()创建，并且CLONE_NEWNS flag置位，进程将得到一个新的namespace。如果CLONE_NEWNS flag没有置位，进程将继承父进程的namespace。

当一个进程安装或卸文件系统，只会修改它的namespace。所以，改动只会对共享同一namespace的进程所见。一个进程甚至可以通过系统调用改变它的root文件系统。

进程的namespace主要数据结构是 namespace。task_struct->namespace指向它。

namespace->list是一个链表头,所有属于该namespace的安装的文件系统挂在该链表中。namespace->root代表着这个namespace的root文件系统。

12.4.2. Filesystem Mounting

Linux中，同种文件系统设备能被安装多次,但是实际上它是唯一的,只是可以通过多个安装点访问它。无论安装多少次，内存中只有一个对应的超级块对象。

每次进行一次安装文件系统的操作,内核必须在内存中保存安装点和mount flags,以及各个文件系统之间的关系。这些信息保存在mounted filesystem descriptor中，数据结构为vfsmount。

vfsmount数据结构存在于几个链表中：

1. 一个hash表，使用父文件系统的vfsmount描述符地址和安装点的dentry对象地址来索引hash表。Hash表保存在mount_hashtable数组中,数组成员是list_head结构。通过vfsmount->mnt_hash挂入hash表中。
2. 对每个namespace,它包含了属于这个namespace的安装的文件系统描述符,每个安装的文件系统描述符通过vfsmount->mnt_list挂入namespace->list链表中。
3. 对每个安装的文件系统,它有一个包含了所有子安装的文件系统的链表。通过vfsmount->mnt_child链入父mounted filesystem descriptor的vfsmount->mnt_mounts。

12.4.3. Mounting a Generic Filesystem

sys_mount()函数请求了big kernel lock,代用do_mount()函数。一旦do_mount()函数返回,将释放big kernel lock。

do_mount()函数执行下列步骤：

1. 如果参数flags中MS_NOSUID, MS_NODEV, 或 MS_NOEXEC设置了,函数清除他们，相应的设置MNT_NOSUID, MNT_NODEV, MNT_NOEXEC flag,结果保存在mnt_flags局部变量中。
2. 调用path_lookup()查询安装点的路径名;函数把搜索结果保存在局部变量nd中，它的数据结构是nameidata。
3. 检测参数flags来决定将执行什么。特别的：

a.如果MS_REMOUNT被设置,说明函数的目的是改变超级块对象的s_flags和mounted filesystem descriptor的mnt_flags。

b.否则,它检测MS_BIND flag。如果设置了,说明需要使一个文件或目录在另一个系统目录树的点可见。

c.否则,它检测MS_MOVE flag。如果设置了,说明需要改变已经安装的文件系统的安装点。do_move_mount()函数将被调用。

d.否则,它调用do_new_mount()。这是最通常的情况。当用户请求安装储存在磁盘分区上的一个特殊文件系统或一个常规文件系统时，该函数被触发。do_new_mount()调用do_kern_mount()函数,传递给它参数：文件系统类型，mount flags和块设备名。do_kern_mount()执行了真正的mount操作并且返回新安装的文件系统描述符的地址。接着， do_new_mount()调用do_add_mount(),do_add_mount()执行了下列动作：

1.请求当前进程的namespace->sem 读/写semaphore,因为函数将修改namespace。

2. do_add_mount()函数可能会使当前进程进入睡眠;在当前进程睡眠时,另一个进程可能在同一个安装点安装文件系统或者甚至改变我们的root文件系统(current->namespace->root)。因此需要确认在这个安装点最后一次安装的文件系统仍然被current的namespace引用;如果不是,释放读/写semaphore并且返回一个错误码。

3.如果被安装的文件系统已经安装在被系统调用参数指定的安装点处,或者安装点是一个symbolic链接,那么将释放读/写semaphore并且返回错误码。

4.初始化新安装的文件系统对象(vfsmount)的mnt_flags。

5.调用graft_tree()把新安装的文件系统对象插入namespace链表,hash表和父安装文件系统的孩子链表。

6.释放nanmespace->sem读/写semaphore并且返回。

4. 调用path_release()结束安装点的路径名查找。

12.4.3.1. The do_kern_mount( ) function

mount操作的核心是do_kern_mount()函数,它检测文件系统的typy flags来决定mount操作如何进行。函数接收下列参数:

1. fstype:要被安装的文件系统类型名。
2. flags: mount flags。
3. name:存储文件系统的块设备的文件路径名(或特殊文件系统的文件系统名)。
4. data:指向传递给文件系统的read_super方法的数据。

函数执行下列步骤:

1. 调用get_fs_type()搜索文件系统类型链表并且定位要安装的文件系统类型,返回file_system_type描述符的地址,存放在type局部变量中。
2. 调用alloc_vfsmnt()分配一个新的mounted filesystem descriptor并且把它的地址存放在mnt局部变量中。
3. 调用type->get_sb()分配一个新超级块并初始化它。
4. 用新分配的超级块初始化mnt->mnt_sb。
5. 用文件系统的root目录的dentry地址初始化mnt->mnt_root,并且递增dentry对象的使用计数。
6. 用mnt的值初始化mnt->mnt_parent(对通用文件系统, mnt_parent的合适的值实在graft_tree()函数中设定)。
7. 用current->namespace初始化mnt->mnt_namespace。
8. 释放超级块的s_umount 读/写semaphore。
9. 返回mnt。

12.4.3.2. Allocating a superblock object

文件系统对象(vfsmount)的get_sb方法在Ext2文件系统中如下实现:

```
struct super_block * ext2_get_sb(struct file_system_type *type,
                            int flags, const char *dev_name, void
*data)
{
        return get_sb_bdev(type, flags, dev_name, data,
ext2_fill_super);
}
```

get_sb_bdev()VFS函数为基于磁盘的文件系统分配并且初始化一个新的超级块;它接收ext2_fill_super()函数的地址, ext2_fill_super()函数从Ext2磁盘分区读取超级块。
VFS为特殊文件系统提供了合适的函数来分配超级块:get_sb_pseudo()函数(没有安装点的特殊文件系统,如pipefs),get_sb_single()(但已安装点的特殊文件系统,如sysfs),和get_sb_nodev()(能够被安装多次的文件系统,如tmpfs)。

get_sb_bdev()执行的最重要的操作有:
1. 调用open_bdev_excl()打开名字为dev_name的块设备。
2. 调用sget()来搜索文件系统的超级块链表(file_system_type->fs_supers)。如果对应于块设备的超级块已经有了,函数返回它的地址。否则,分配并初始化一个新超级块对象,插入到文件系统的超级块链表和全局的超级块链表,并返回它的地址。
3. 如果超级块不是新的(说明文件系统已经被安装了),跳转到6。
4. 拷贝flags参数内容到超级块的s_flags域并且根据块设备恰当的设置s_id,s_old_blocksize和s_blocksize域的值。
5. 调用作为参数传进来的函数指针访问磁盘上的超级块信息并填充分配的超级块。
6. 返回新超级块对象地址。

12.4.4. Mounting the Root Filesystem

安装root文件系统是分两个阶段:
1. 内核安装特殊文件系统rootfs,它简单的提供了一个空目录作为初始的安装点。
2. 内核在这个空目录上安装真正的root文件系统。

内核安装roofs文件系统是为了使真正的root文件系统容易变更。事实上,在一些情况下,内核安装和卸载许多root文件系统,一个接着一个。例如,一个内核发布版的启动CD会加载一个只有很少驱动的内核进RAM,它加载一个很小的文件系统进ramdisk作为root。接着,在初始root文件系统里的程序探测系统硬件(例如,决定磁盘是SCSI,IDE或其他),加载内核所需的模块,并且从物理块设备上重新安装root文件系统。

12.4.5. Unmounting a Filesystem

sys_umount()接收两个参数:文件名(要么是安装点目录要么是块设备文件名)和flags集合。函数执行下列动作:
1. 调用path_lookup()来查询安装点路径名;这个函数返回查询结果在nameidata类型的局部变量nd中。
2. 如果找到的目录不是一个文件系统的安装点,设置retval返回错误码-EINVAL并跳转到6。这个检测是通过确认nd.mnt->mnt_root是否等于nd.dentry来实现。
3. 如果被卸载的文件系统还没有安装到namespace中,函数设置retval为-EINVAL并且跳转到6(一些特殊文件系统没有安装点)。这个检测是通过调用check_mnt(nd.mnt)。
4. 如果用户没有卸载文件系统的权限,函数设置retval为-EPERM并且跳转到6。
5. 调用do_umount(),参数为nd.mnt,flags。函数主要执行下列操作:
    a. 从nd.mnt->mnt_sb得到超级块地址放入sb中。
    b. 如果用户要求强制执行卸载操作,它通过执行umount_begin(),将打断正在进行的安装操作。
    c. 如果被卸载的文件系统是root文件系统并且用户没有要求真正的卸载它,函数调用哪个do_remount_sb()重安装root文件系统为只读,并结束。

d. 请求为当前进程的namespace->sem读/写semaphore的写者，得到vfsmount_lock spin lock。
e. 如果文件系统有包含孩子文件系统，则调用umount_tree()卸载文件系统（包括孩子文件系统）。
f. 释放vfsmount_lock spin lock和amespace->sem。

6. 减少对应与卸载的文件系统的root目录的dentry使用计数和vfsmount使用计数，这些计数器通过path_lookup()增加。
7. 返回retval值。

12.5. Pathname Lookup

路径查找的函数为path_lookup()，它有3个参数：
name:指向一个要解析的路径名。
flags: The value of flags that represent how the looked-up file is going to be accessed.
nd:nameidata数据结构，保存了查询操作的结果。
当path_lookup()返回时，nameidata结构保存了相关数据。

nameidata结构中，dentry和mnt成员分别代表了在路径名中最后一个被解析的部分的dentry对象和vfsmount对象。这两个成员描述了给定路径名指定的文件。

path_lookup()函数执行了下列步骤：
1. 初始化参数nd的一些成员：
a. 设置nameidata->last_type为LAST_ROOT(如果路径名是一个slash或者一连串slash，它将被使用)。
b. 设置nameidata->flags为参数flags。
c. 设置nameidata->depth为0。
2. 请求当前进程的current->fs->lock读/写semaphore为读者。
3. 如果路径名的第一个字符是slash，那么查询操作必须从当前进程的root目录开始：函数把current->fs->rootmnt和current->fs->root分别存放在nd->mnt，nd->dentry中。
4. 否则，如果第一个字符不是slash，查询操作必须从当前进程的工作目录开始:函数把current->fs->pwdmnt和current->fs->pwd分别存放在nd->mnt，nd->dentry中。
5. 释放当前进程的current->fs->lock读/写semaphore。
6. 设置task_struct->total_link_count为0。
7. 调用link_path_walk()函数。
```
return link_path_walk(name, nd);
```

12.5.1. Standard Pathname Lookup

当LOOKUP_PARENT标志位被清空， link_path_walk()执行下列步骤：
1. 用nd->flags初始化lookup_flags局部变量。
2. 跳过路径名中第一个组件前面的所有slashes。
3. 如果剩下的路径名为空，返回0。在nameidata结构中，dentry和mnt指向了与原始路径名最后一个组件相关的对象。
4. 如果nd->depth为正，设置lookup_flags局部变量中的LOOKUP_FOLLOW标志。
5. 执行循环不断的把从name传进来的路径名分割成一些组件；对每个组件：
a. 从nd->dentry->d_inode中找到最后一个解析出来的组件的inode对象地址。

b. 检查最后一个解析出来的组件的inode对象中的权限。如果inode有一个专用的permission方法，函数将执行它；否则，它执行exec_permission_lite()函数，这个函数将检测保存在inode->i_mode中的访问权限和当前进程的特权级。如果最后一个解析出来的组件不允许执行，link_path_walk()跳出循环并返回一个错误码。
c. 考虑下一个要解析的组件。从这个名字中函数计算32-bit的hash值用来在dentry cache hash table中查找它的dentry对象。
d. 当解析完一个组件后跳过所有结尾的slash。
e. 如果被解析的是路径名中最后一个组件，跳转到步骤6。
f. 如果组件名是"."，函数将继续解析下一个组件。
g. 如果组件名是".."，它尝试climb to父目录。

如果最后一个解析出来的目录是进程的root目录，那么climbing是不允许的：函数调用follow_mount()并且继续解析下一个组件。

如果最后一个解析出来的目录是nd->mnt文件系统的root目录并且nd->mnt文件系统没有安装在另一个文件系统中(nd->mnt等于nd->mnt->mnt_parent)，那么nd->mnt文件系统通常是namespace的root文件系统：在这种情况下，climbing是不可能的，那么调用follow_mount()并且继续解析下一个组件。

如果最后解析出来的目录是nd->mnt文件系统的root目录，并且nd->mnt文件系统安装在另一个文件系统上，那么需要文件系统的切换。因此，函数设置nd->dentry为nd->mnt->mnt_mountpoint，设置nd->mnt为nd->mnt->mnt_parent，跳转到5g。

如果最后一个解析出的目录不是文件系统的root目录，那么函数必须简单的climb to父目录：设置nd->dentry为nd->dentry->d_parent，调用follow_mount()并继续解析下一个组件。

follow_mount()函数检测nd->dentry是否是一个安装点(nd->dentry->d_mounted大于0)；如果是，调用lookup_mnt()在dentry cache中查找安装在这个文件系统的root目录，并且更新nd->dentry和nd->mnt。接着重复整个操作。

h. 如果组件名既不是"."也不是"..."，那么函数必须在dentry cache中查找。如果文件系统有专用的d_hash方法，函数将调用它来修改5c中计算出来的值。
i. 设置nd->flags的LOOKUP_CONTINUE标志位，代表存在下一个组件需要解析。
j. 调用do_lookup()，寻找要解析的组件的dentry对象。do_lookup()本质上是调用__d_lookup()首先在dentry cache中寻找dentry对象。如果不存在，do_lookup()调用real_lookup()，这个函数执行inode的lookup方法从磁盘中读取目录，创建一个新的dentry对象并插入dentry cache，接着创建一个新的inode对象并插入inode cache。这个步骤结束时，next局部变量的dentry和mnt成员指向了要解析的组件的相关对象。
k. 调用follow_mount()函数检查刚被解析的组件是否是一个安装点。如果是，follow_mount()更新next.dentry和next.mnt使他们指向安装点中最上层的文件系统的root目录和vfsmount。
l. 检查刚被解析的组件是否是一个symbolic link(next.dentry->d_inode有一个专用的lookup方法)。
m. 检查刚被解析的组件引用的是一个目录(next.dentry->d_inode有一个专用的lookup方法)，如果不是，返回错误码-ENOTDIR，因为组件是在路径名的中间。
n. 设置nd->dentry为next.dentry，设置nd-mnt为next.mnt，接着解析路径名中的下一个组件。

6. 现在，所有的原始路径名中的组件，除了最后一个，都被解析了。清除nd->flags中的LOOKUP_CONTINUE标志。

7. 如果路径名有一个slash作为结尾，设置lookup_flags局部变量的LOOKUP_FOLLOW和LOOKUP_DIRECTORY标志位，迫使最后一个组件被解释为目录名。
8. 检测lookup_flags是否有LOOKUP_PARENT标志位。我们假设这个标志位为0。
9. 如果最后一个组件是”.”，结束执行并返回0。在nd中，dentry和mnt指向的是”.”之前的一个组件的相关对象。
10. 如果最后一个组件是”..”，那么要尝试climb to父目录。
11. 最后一个组件既不是”.”也不是”..”，因此函数必须在dentry cache中查找。如果文件系统提供了专用d_hash方法，调用它，替换5c中计算出的hash值。
12. 调用do_lookup()得到要解析的组件的dentry对象。
13. 调用follow_mount()检查最后一个组件是否是安装点。如果是，则要接着往下查询到最后一个文件系统的root目录。
14. 检查lookup_flags中LOOKUP_FOLLOW是否被设置了并且next.dentry->d_inode含有一个专用follow_link方法。如果都满足，组件是一个symbolic link的话，就要被follow_link方法解释。
15. 如果组件不是symbolic link，或symbolic link不应该被解释，则分别设置nd->mnt和nd->dentry为next.mnt和next.dentry。nd此时包含了查询的最终结果。
16. 检测是否nd->dentry->d_inode为NULL。如果为NULL，通常是因为路径名关联到了一个不存在文件。在这种情况下，函数返回错误码-ENOENT。
17. 现在有一个inode与查询结果关联。如果LOOKUP_DIRECTORY置位，检查inode是否有专用lookup方法，如果有，它是一个目录。否则，函数返回错误码-ENOTDIR。
18. 返回0。nd->dentry和nd->mnt指向了路径名的最后一个组件。

12.5.2. Parent Pathname Lookup

如果查询操作需要解析的包含路径名中最后一个组件的目录，而不是路径名中最后一个组件，那么需要在参数flags中设置LOOKUP_PARENT标志位。

当LOOKUP_PARENT标志位被设置了，link_path_walk()函数最后会设置nameidata的last和last_type成员。last成员保存了路径名中最后一个组件的名字。last_type指明了最后一个组件的类型。

与上一节查询操作不同之处在于：
1. 设置nd->last为最后一个组件的名字。
2. 初始化nd->last_type为LAST_NORM。
3. 如果最后一个组件的名字是”.”，那么设置nd->last_type为LAST_DOT。
4. 如果最后一个组件的名字是”..”，那么设置nd->last_type我哦哦LAST_DOTDOT。

当函数结束时，最后一个组件完全没有被解析，nameidata的dentry和mnt成员指向包含最后一个组件的目录的相关对象。

12.5.3. Lookup of Symbolic Links

一个symbolic link是一个常规文件，它保存了另一个文件的路径名。

symbolic link可能会嵌套，因此task_struct->link_count记录了查询操作递归层次，最多5层，第6层就会出错返回。而且， task_struct->total_link_count记录了symbolic link总数，当该值到达了40，查询操作就会出错返回。

link_path_walk()函数调用do_follow_link(),参数为symbolic link的dentry对象和nameidata数据结构地址。do_follow_link()执行下列步骤:

1. 检查current->link_count是否小于等于5,否则出错返回-ELOOP。
2. 检查current->total_link_count小于;否则出错返回-ELOOP。
3. 调用cond_resched(),如果当前进程的TIF_NEED_RESCHED置位,将执行一次进程切换。
4. 递增current->link_count,current->total_link_count,和nd->depth。
5. 更新与symbolic link文件相关的inode对象的访问时间。
6. 调用依赖于文件系统的follow_link()方法,它将从symbolic link的inode中提取路径名,并且把它保存在nd->saved_names数组中。
7. 调用__vfs_follow_link()函数。
8. 如果定义了,执行inode->put_link函数,它将释放follow_link方法临时分配的数据结构。
9. 递减current->link_count好nd->depth。
10. 返回__vfs_follow_link()函数的返回值。

12.6. Implementations of VFS System Calls

12.6.1. The open( ) System Call

sys_open()函数执行下列步骤:

1. 调用getname()从进程地址空间读取文件路径名。
2. 调用get_unused_fd()在current->files->fd中找到一个空槽。相应的index保存在fd局部变量中。
3. 调用filp_open()函数,传递参数:路径名,访问模式标志,权限bit mask。这个函数执行下列步骤:

a. 拷贝访问模式标志到namei_flags中,但是对O_RDONLY, O_WRONLY, and O_RDWR要进行编码,编码格式是:当文件访问需要读权限时,namei_flags的bit0置位;当文件访问需要写权限时,bit1置位。注意到,不可能指定open()系统调用访问文件既不需要读权限也不需要写权限。然而,这对包含symbolic links的路径查找操作是有意义的。

b. 调用open_namei(),传递参数:路径名,修改后的访问模式标志namei_flags和nameidata类型的局部变量地址。函数以下列方式执行查询操作:
   - 如果O_CREAT没有置位,查询操作中LOOKUP_PARENT清0,LOOKUP_OPEN置位。而且,仅当O_NOFOLLOW清0时才给LOOKUP_FOLLOW置位,仅当O_DIRECTORY置位时才给LOOKUP_DIRECTORY置位。
   - 如果O_CREAT置位,查询操作中LOOKUP_PARENT, LOOKUP_OPEN, LOOKUP_CREATE置位。一旦path_lookup()函数成功返回,检查请求的文件是否存在。如果不存在,分配一个新的磁盘inode。

c. 调用dentry_open()函数,参数为:查询中找到的dentry对象, 查询中找到的vfsmount,访问模式标志。这个函数执行下列步骤:

- 分配一个新的file对象。
- 根据open()系统调用传入的文件访问权限初始化file对象的f_flags和f_mode成员。
- 根据参数初始化f_dentry和f_vfsmnt成员。
- 设置f_op成员为相应inode->i_fop。这一步设置好了用于将来文件操作的所有方法。
- 把文件对象插入文件系统的超级块的s_files链表。

- 如果file对象的open方法定义了,执行它。
- 调用file_ra_state_init()来初始化read-ahead数据结构。
- 如果O_DIRECT置位,检查是否能在文件上执行直接I/O操作。
- 返回file对象的地址。

d. 返回file对象的地址。
4. 设置current->files->fd[fd]为dentry_open()返回的文件对象的地址。
5. 返回fd。

12.6.2. The read( ) and write( ) System Calls

sys_read()和sys_write()执行相似的步骤:
1. 调用fget_light()取得file对象的地址。
2. 如果file->f_mode指明不允许访问,返回错误码-EBADF。
3. 如果file对象没有read()或aio_read()方法（write()，aio_write()）,返回错误码-EINVAL。
4. 调用access_ok()检测buf和count参数的合法性。
5. 调用rw_verify_area()检查是否有强制锁在要访问的区域，如果是,返回一个错误码,或者如果锁是用一个F_SETLKW命令请求的,则使当前进程进入睡眠。
6. 如果定义了,要么调用file->f_op->read要么调用file->f_op->write方法来传输数据。否则， 要么调用 file->f_op->aio_read要么调用file->f_op->aio_write方法来传输数据。这些方法,返回真实传输的字节数,同时file pointer也会更新。
7. 调用fput_light()释放文件对象。
8. 返回真正传输的字节数。

12.6.3. The close( ) System Call

sys_close()执行下列步骤:
1. 从current->files->fd[fd]中得到file对象地址;如果是NULL,返回一个错误码。
2. 设置current->files->fd[fd]为NULL。通过清除current->files的open_fds和close_on_exec成员的相应bit来释放file描述符fd。
3. 调用filp_close(),它执行下列操作:

a. 如果有定义,调用file对象的flush方法。
b. 释放文件中的所有强制锁。
c. 调用fput()释放file对象。

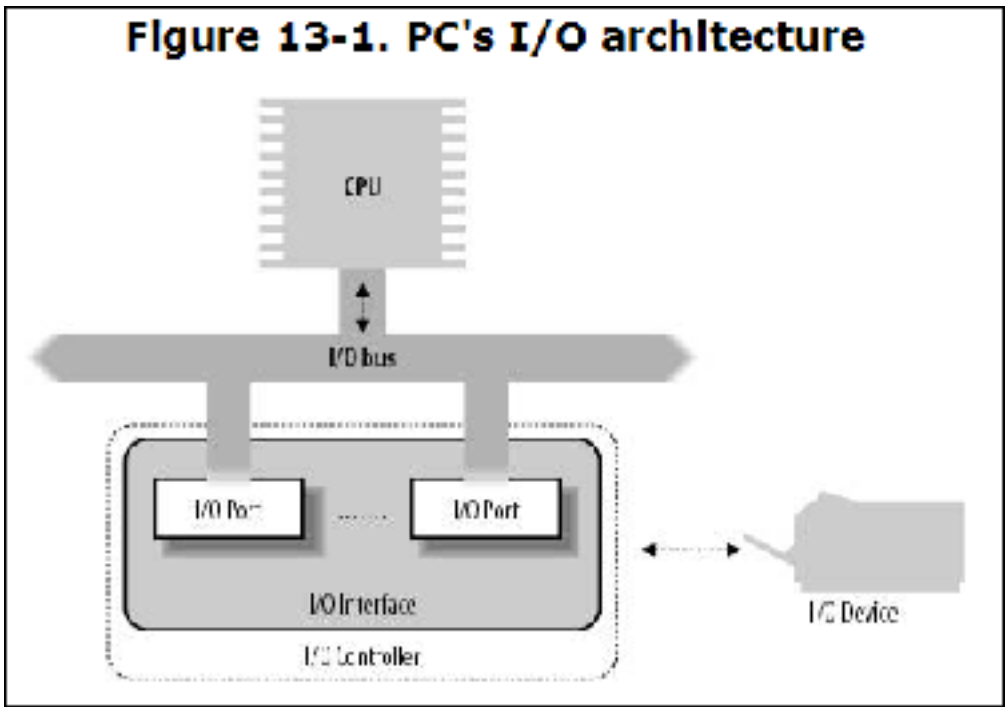
4. 返回0或一个错误码。

12.7. File Locking

Chapter 13. I/O Architecture and Device Drivers

13.1. I/O Architecture

I/O ports , interfaces, and device controllers



13.1.1. I/O Ports

每个连接到I/O bus上设备有自己的I/O地址集合,通常叫做I/O ports。

内核跟踪分配给每个硬件设备的I/O ports，其方法称为"resources"。

一个resource代表了一些只能独占的分配给设备驱动的实体的一部分。在我们这里,一个resource代表了提个I/O port地址范围。与每个resource相关的信息保存在resource数据结构中。同种类型的所有resource被插入一个tree-like数据结构;例如,所有的代表I/O port 地址范围的resources被包含在一棵树中，树根保存在ioport_resource中。

13.1.2. I/O Interfaces

一个I/O interface 是一个硬件电路,它在一组I/O ports和对应的device controller之间。它表现为一个解释器,把I/O ports的值解释为命令和数据。在相反地方向上,它检测到设备状态的变化并相应的更新用作状态寄存器的I/O port。这个电路也能通过IRQ线连接到一个Programmable Interrupt Controller,因此它代替设备发送一个中断请求。

有两种interface:

1. Custom I/O interfaces: Keyboard interface, Graphic interface, Disk interface, Bus mouse interface, Network interface。
2. General-purpose I/O interfaces: Parallel port, Serial port, PCMCIA interface, SCSI, Universal serial bus (USB)。

13.1.3. Device Controllers

device controller有两个重要的角色:

1. 它把从I/O inteface接收到的高级命令解释成发给设备的电信号。
2. 它解释和转换从设备接收到的电信号并且修改状态寄存器。

13.2. The Device Driver Model

13.2.1. The sysfs Filesystem

sysfs文件系统的目标是展现device driver model中组件的层次关系。这个文件系统中最顶层的是：

1. block:块设备，独立于它们连接到的bus。
2. devices: 所有被内核识别的硬件设备，根据连接到的bus来进行组织。
3. bus:系统bus。
4. drivers:内核中注册的设备驱动。
5. class:系统中的设备类型。
6. power:处理一些硬件设备的power状态的文件。
7. firmware: 处理一些硬件设备的firmware的文件。

device driver model中组件的关系在sysfs文件系统中以目录和文件之间的symbolic link来表现。

sysfs文件系统中常规文件的主要角色是代表驱动和设备的属性。

13.2.2. Kobjects

device driver model的核心数据结构是一个名为kobject的通用数据结构，每个kobject对应sysfs文件系统中的一个目录。

Kobject被嵌入一个大的对象，被称为"containers"，它描述了device driver model中的组件。Buses，devices和drivers的描述符是典型的containers的例子。在container中内嵌的kobject允许内核：

1. 为container维护一个引用计数。
2. 维护层次链表或container的集合。
3. Provide a User Mode view for the attributes of the container。

13.2.2.1. Kobjects, ksets, and subsystems

kobject数据结构的成员：

1. k_name:指向一个string，保存了container的名字。
2. name: 如果container的名字在20字节以内，就存在这。
3. kref:container的引用计数。
4. entry:kobject通过它来链入别的链表。
5. parent:指向父kobject。
6. kset:指向包含的kset。
7. ktype:指向kobject类型描述符。
8. dentry:指向与这个kobject相关的sysfs文件的dentry对象。

ktype成员指向一个kobj_type对象，代表了kobject的"type"，本质上是包含该kobject的container的类型。kobj_type数据结构包含3个成员：一个release方法(当kobject被释放时执行)，一个sysfs_ops指针，指向一个sysfs操作表，和一个sysfs文件系统的缺省属性链表。

kref成员是一个kref类型的结构，它仅有一个refcount成员。正如它的名字暗示的，这个成员是kobject的引用计数，同时它也表现为包含这个kobject的container的引用计数。kobject_get()和kobject_put()函数递增和递减引用计数。如果计数器变成0，那么被kobject使用的资源被释放，kobj_type->release方法被执行。这个方法通常仅在kobject所属的container是动态分配时被定义，它释放container。

kobjects 能通过ksets组织成一个层次树。一个kset是一个同类型的kobjects（包含在同类型的container中）集合体。kset数据结构的成员是:

1. subsys:指向subsystem描述符。
2. ktype:指向kset的kobject类型描述符。
3. list: 包含在kset中的kobject链表头。
4. kobject:内嵌的kobject。
5. hotplug_ops:指向用于kobject过滤和热拔插的回调函数表。

kset->ktype指向kset中的所有kobjects共享的kobj_type描述符。

kset->kobj是kset数据结构中一个内嵌的kobject; kset->kobj->parent指向这个内嵌的kobject。因此,一个kset是一个kobject的集合体,但是它依赖于更高层的kobject来引用计数和链入分层树。例如,kset_get()和kset_put()函数,分别是增加和减少kset的引用计数,它们简单的调用kobject_get()和kobject_put()作用在内嵌的kobject上;因为kset的引用计数仅仅只是kset中内嵌的kobject的引用计数。此外,因为有内嵌的kobject,kset能被内嵌进一个"container"对象,就像kobject数据结构那样。最后,一个kset能成为另一个kset的成员:它能把内嵌的kobject插入一个更高层次的kset。

kset的集合体成为subsystem。一个subsystem可能包含不同类型的kset,并且它的的类型subsystem数据结构只有2个成员:
kset:一个内嵌的kset。
rwsem:一个read-write semaphore保护迭代的包含在subsystem中的所有ksets和kobjects。

甚至连subsystem也能内嵌进一个更大的"container"对象;container的引用计数就是内嵌的subsystem的引用计数…最后归结为kobject的引用计数。subsys_get()和subsys_put()函数用来增加和减少引用计数。

13.2.2.2. Registering kobjects, ksets, and subsystems

kobject_register()初始化了一个kobject并且在sysfs文件系统中添加了相应的目录。在调用它之前,调用者应该把kobject->set设置为指向其父kset。
kobject_unregister()从sysfs文件系统中移除kobject的目录。内核还提供了kset_register()和kset_unregister(),subsystem_register()和subsystem_unregister(),但是他们本质上都是kobject_register()和kobject_unregister()的包装函数。

许多kobject目录包含了名为attributes的常规文件。sysfs_create_file()函数接收参数:一个kobject的地址和一个attribute描述符,然后在合适的目录中创建文件。另一种objects之间的关系在sysfs文件系统中用symbolic link来代表:
sysfs_create_link()函数为一个给定的kobject到另一个kobject创建了一个symbolic link。

13.2.3. Components of the Device Driver Model

device driver model是建立在几个基本的数据结构之上的,代表了buses,devices,device drivers,etc。

13.2.3.1. Devices

每个设备在device driver model中使用一个device对象代表着。

device对象被全局变量struct subsystem devices_subsys所收集,属于/sys/devices目录下。设备被分层组织:一个设备是一些"孩子"设备的"父亲",说明"孩子"设备离开了"父亲"设备就不能正常的工作。device->parent指向它的父亲设备描述符,通过device->node链入父亲device->children链表。device对象中内嵌的kobject的父子关系也反映了设备的层次关系;因而,/sys/devives下的目录结构与硬件设备的物理组织匹配。

每个驱动维护了一个它管理的设备链表;设备通过device->driver_list链入驱动对象,同时, device->driver指向设备驱动描述符。对每一个bus类型,有一个链表包含了所有插在该bus上面的设备; 设备通过device->bus_list链入bus对象,同时device->bus指向bus类型描述符。

device对象的使用的引用计数是由其内嵌的kobject维护。

device_register()函数把一个新的device对象插入device driver model,并且自动的在/sys/devices/下创建一个目录。device_unregister()函数执行相反的操作。

通常,device对象静态的内嵌在一个大的描述符中。例如,PCI设备使用pci_dev数据结构描述; pci_dev->dev就是一个device对象,其它成员是属PCI bus专用。

13.2.3.2. Drivers

每个驱动在device driver model使用device_driver对象描述。

device_driver对象包含了4个方法用于处理热拔插,即插即用和电源管理。probe方法当一个设备驱动发现一个设备可能可以被驱动处理时调用;相应函数会探测硬件并对设备执行进一步检测。remove方法当一个可热拔插设备被移除时调用;当驱动自己被移除时,他也会被调用来处理每个设备。shutdown, suspend,和resume方法用于内核改变电源状态。

通常,devic_driver对象静态的被内嵌在一个更大的描述符中。例如,PCI设备驱动使用pci_driver数据结构描述; pci_driver->driver就是个devic_driver对象,其它部分都是PCI bus专用成员。

13.2.3.3. Buses

每个bus类型被内核用bus_type对象描述。

每个bus_type对象包含一个内嵌的subsystem;全局变量bus_subsys(我在内核源代码里没有找到对应声明,奇怪)收集了所有内嵌在bus_type对象中subsystem。bus_subsys subsystem与/sys/bus目录相联系。The per-bus subsystem典型的包含了2个ksets:drivers和devices。(分别对应与bus_type对象的drivers和devices成员)

13.2.3.4. Classes

每个class使用一个class对象描述。所有的class对象属于class_subsys subsystem,它与/sys/class目录相联系。每个class对象包含一个内嵌的subsystem。

每个class对象包含了一个class_devices描述符链表,其中每个描述符代表了一个属于该class的单独的逻辑设备。class_devices数据结构中有一个dev成员指向一个device描述符,因而一个逻辑设备总是引用了一个特定的真实设备。

device driver model中的class本质上为了提供一个标准的方法来向用户模式应用提供逻辑设备的接口。每个class_device描述符内嵌一个kobject,它包含名为dev的属性。这个属性保存了设备文件的major和minor号。

13.3. Device Files

设备文件通常在文件系统中储存为一个真正的文件。它的inode,不需要包含真正的指向磁盘数据块的指针,而是必须要包含一个硬件设备的标识符。

内核并不关心设备文件的名字,因此，块设备/tmp/disk 设备号3:0和块设备/dev/had 设备号3:0是等同的。

13.3.1. User Mode Handling of Device Files

13.3.1.1. Dynamic device number assignment

Linux内核能够动态的创建设备文件:没有必要在/dev目录下创建每一个可能的设备文件,因为设备文件能被"on demand"创建。2.6内核在device driver model下能够以很简单的方法实现这一目的。

13.3.2. VFS Handling of Device Files

当设备文件被打开时,VFS改变了设备文件的缺省操作;每个针对设备文件发出的系统调用被转换成一个设备相关的函数调用,而不使用设备文件所在文件系统的函数。设备相关的函数表现为硬件设备执行进程请求的操作。

在一个设备文件上执行open()系统调用,当它的inode对象被创建和初始化后,函数发现它与一个设备文件相关,那么调用init_special_inode(),这个函数将用设备文件中的major和minor号初始化inode对象的i_rdev成员,根据设备文件类型设置inode->i_fop为def_blk_fops或def_chr_fops文件操作表的地址。open()系统调用还会创建一个file对象并设置它的f_op为i_fop的值。从此,接下来对设备文件的系统调用将使用设备驱动的函数而不是使用文件系统的函数。

13.4. Device Drivers

13.4.1. Device Driver Registration

例如:一个通用的PCI设备。为了处理它,它的驱动必须分配一个pci_driver类型的描述符,它将被PCI内核层用来处理该设备。在初始化这个描述符的一些成员后,设备驱动调用pci_register_driver()函数。事实上,pci_driver描述符包含了一个内嵌的device_driver描述符; pci_register_driver()简单的初始化了内嵌的driver描述符后调用driver_register()把driver描述符插入device driver model。

当一个设备驱动被注册后,内核检查之前不支持的设备,现在是否能被该驱动来处理了。为了实现这个目的,它使用bus_type->match方法,然后device_driver->probe方法。如果一个能被驱动处理的硬件设备被发现了,内核分配一个device对象并调用device_register()把设备插入device driver model。

13.4.2. Device Driver Initialization

注册一个设备驱动和初始化设备是两件不同的事情。一个设备驱动可以随时被注册，因此用户模式的应用程序能通过设备文件来使用它。相反的，一个设备只在最后一刻被初始化。事实上，初始化一个设备意味着分配系统资源给它。

为了确保系统资源仅在需要时才被分配，设备驱动使用了如下方案：

1. 一个计数器追踪了目前正在访问设备文件的进程数目。当调用设备文件的open方法时递增，调用release方法时递减。
2. open方法在递增计数器值前检查它的值。如果计数器为0，设备驱动分配资源并且开启中断和DMA。
3. release方法在递减计数器后检查它的值。如果计数器为0，那么没有进程正在使用硬件设备。那么，该方法将关闭该设备中断和DMA，释放分配给它的资源。

13.4.3. Monitoring I/O Operations

有两种技术用来监视I/O操作的结束：polling mode和interrupt mode。

13.4.3.1. Polling mode

如果完成I/O操作所需的时间较长，这个方案将变得效率低下，因为CPU浪费了宝贵的时钟周期在等待I/O完成。在这样的情况下，每次polling操作后主动的放弃CPU是种好方法，可以在循环中插入schedule()函数。

13.4.3.2. Interrupt mode

interrupt mode仅在I/O控制器能够在I/O操作结束时，通过IRQ线发送中断时，才能使用。

13.4.4. Accessing the I/O Shared Memory

根据设备和bus类型，被I/O共享的内存可能被映射到不同的地址范围内。典型的有：
For most devices connected to the ISA bus:
I/O共享内存通常被映射到16-bit物理地址，范围为0xa0000到0xfffff。
For devices connected to the PCI bus:
I/O共享内存被映射到4GB附近的32-bit物理地址。这种类型的设备更容易处理。

内核程序使用的是线性地址，因此I/O共享内存的线性地址必须大于PAGE_OFFSET。设备驱动必须把I/O物理地址转换成内核空间的线性地址。这一步只需要简单的把32-bit物理地址与常量0xc0000000相或。

当I/O物理地址处于高地址空间时，就必须使用ioremap()或ioremap_nocache()进行映射。ioremap()类似于vmalloc()，调用了get_vm_area()来创建一个新的vm_struct描述符，然后更新相应的Page Table entries。ioremap_nocache()不同于ioremap()之处在与它关闭了硬件cache。

13.4.5. Direct Memory Access (DMA)

CPU与DMA同一时刻访问相同的内存区域的冲突已被名为memory arbiter的硬件设备解决。

DMA主要用在磁盘驱动中和其它一次要传输许多字节的设备。因为给setup time for DMA相对较长，当要传输的字节数较少时，直接使用CPU来做数据传输更加有效些。

13.4.5.1. Synchronous and asynchronous DMA

一个设备驱动使用DMA有两种不同的方法:同步DMA和异步DMA。同步DMA的数据传输由进程触发;异步DMA的数据传输由硬件设备触发。

13.4.5.3. Bus addresses

Bus address是被除了CPU的所有硬件设备drive数据总线的内存地址。

X86体系中,bus address与物理地址重合。有些体系结构,例如Sun的SPARC和Hewlett-Packard的 Alpha有一个名为I/O Memory Management Unit(IO-MMU)的硬件电路,它拥有类似于CPU的分页模块的功能。IO-MMU把物理地址映射为总线地址。所有的使用DMA的I/O驱动在开始数据传输前必须建立合适的IO-MMU。

13.4.5.4. Cache coherency

X86体系中,当使用DMA时没有cache coherency问题,因为硬件设备自几将take care of "snooping" the accesses to the hardware caches。

13.5. Character Device Drivers

一个字符设备驱动被一个cdev结构描述。

cdev结构中,list成员是一个链表头,它收集了使用同种字符设备驱动的所有字符设备文件的inode。可能有很多设备文件有相同的设备号,它们都是引用相同的字符设备。而且,一个设备驱动能够与一个设备号范围相联系,不仅仅是一个单一的设备号;所有设备文件的设备号落在这个范围内的,都会被同样的字符设备驱动处理。设备号范围大小被保存在count成员里(我觉的major是相同的,范围指的是minor号)。
cdev_alloc()函数动态的分配一个cdev描述符,初始化描述符内嵌的kobject。因此,cdev描述符当引用计数为0时,能够自动的被释放。

cdev_add()函数在device driver model中注册了一个cdev描述符。函数初始化了cdev描述符中的dev和count成员。然后调用kobj_map()函数。这个函数建立device driver model的数据结构,数据结构将把设备号范围和设备驱动联系起来。

device driver model为字符设备定义了一个kobject mapping domain,它由类型为kobj_map的描述符代表并被cdev_map全局变量引用。kobj_map描述符包含了一个大小为255的hash表,使用major号索引。Hash表保存了类型为probe的对象,每个对象对应一个注册了的major和minor号范围。

当调用kobj_map()函数时,特定的设备号间隔被加入hash表。对应的probe->data指向了设备驱动的cdev描述符。probe->data会被传递给get和lock方法。

kobj_lookup()函数接收参数: 一个kobject mapping domain和一个设备号;它搜索hash表并returns the address of the kobject of the owner of the interval including the number。这里我们用的是字符设备,函数返回cdev描述符中内嵌的kobject地址。

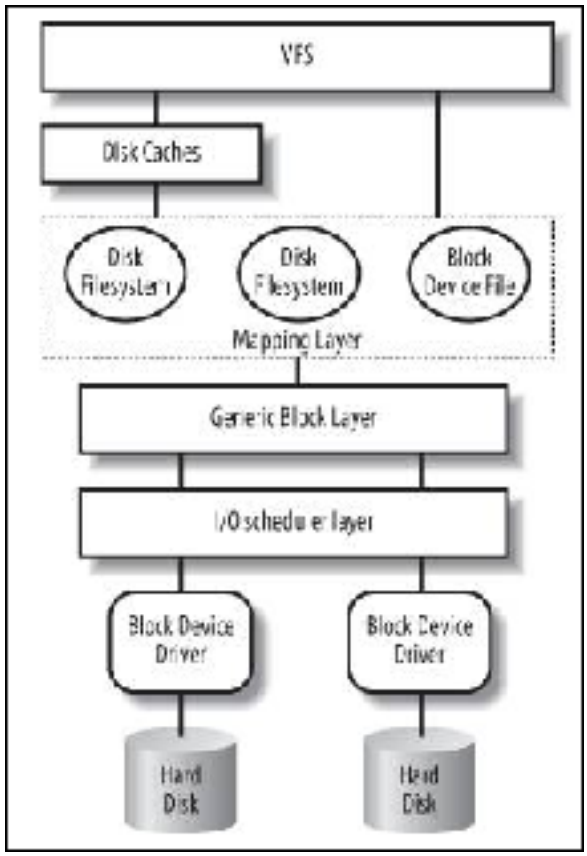
13.5.1. Assigning Device Numbers

为了追踪当前哪些字符设备号被分配了,内核使用一个hash表chrdevs,它包含了设备号范围。2个范围可能会使用相同的major号,但它们不能有重叠,那么它们的minor号必须不同。表有255个entries,hash函数屏蔽了major号的最高4位,major号小于255的将被哈希到不同的entries。每个entry指向冲突链表的第一个元素,链表以major和minor号递增顺序排序。

每个链表元素是char_device_struct结构。

Chapter 14. Block Device Drivers

14.1. Block Devices Handling



当一个进程发出read()系统调用在磁盘文件上,内核对进程的响应是:

1. read()系统调用的服务程序激活一个合适的VFS函数,传递给它一个文件描述符和一个文件内偏移。
2. VFS函数决定如果请求的数据已经可用了,如何执行读操作。有时不需要访问磁盘上的数据,因为内核在RAM中保存最近从磁盘中读取过的数据。
3. 假设内核必须从块设备中读取数据,那么它必须决定数据的物理位置。为了实现该目的,内核依靠mapping layer,它将执行这两步:
   a. 它决定文件系统的块大小和计算请求的数据所占的块数量。本质上,文件被看作被分割成很多块,内核第几块(相对于文件开始)包含了请求的数据。

b. 接着，mapping layer调用了一个特定于文件系统的函数来访问文件的磁盘上的inode，然后决定请求数据在磁盘上的位置，以逻辑块号为单位。本质上，磁盘被看为被分割成块，内核来计算保存请求数据的块号(相对于磁盘或分区的开始)。因为一个文件可能保存在磁盘上不连续的块中，保存在磁盘inode中的一个数据结构把文件块号映射成逻辑块号。

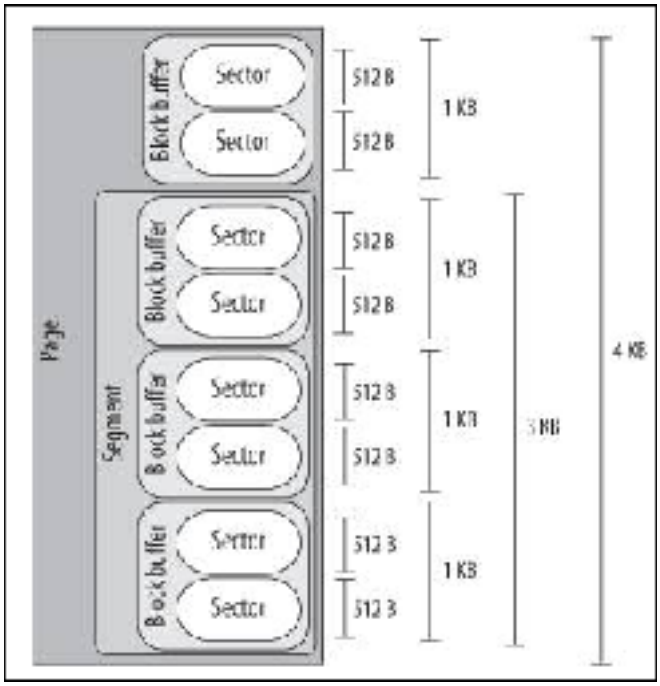
[*]如果读操作访问的是一个raw block device file，mapping layer不会调用特定于文件系统的方法；它把块设备中的偏移转换为磁盘中的对应位置。

4. 内核现在能在块设备上发出读操作了。它利用了generic block layer。generic block layer传输请求的数据。通常，每个I/O操作包含了一组磁盘上相邻的块。因为请求的数据并不强制在磁盘上相邻，generic block layer可以开始许多I/O操作。每个I/O操作被一个"blioc I/O"结构代替，它收集了所有被更低层的组件所需的信息，用来满足读请求。
   generic block layer隐藏了每个硬件块设备的特性，因而提供了一个块设备的抽象。因为绝大多数块设备是磁盘，generic block layer也提供了一些通用的数据结构来描述磁盘和磁盘分区。
5. 在generic block layer之下，"I/O schedule"根据预先定义的内核策略把pending I/O数据传输请求排序。Schedule的目的是为了把数据组织成在物理介质上邻近。
6. 最后，block device driver通过发送合适的命令给磁盘控制器硬件接口，完成了实际的数据传输。

正如所看到的，有很多内核组件与保存在块设备上的数据相关；每个组件使用不同长度的chunks来管理磁盘数据：

1. 块设备硬件控制器使用固定大小的chunks传输数据，被称为"sector"。所以，I/O schedule和块设备驱动必须以sector为单位管理数据。
2. 虚拟文件系统，mapping layer和文件系统以名为"block"的逻辑单元组织磁盘数据。一个block在文件系统中对应最小的存储单元。
3. 块设备驱动可以应对"segment"为单位的数据：每个segment是一个内存页面或内存页面的部分，它包含了磁盘中物理上邻近的多个chunk的数据。
4. The disk caches work on "pages" of disk data, each of which fits in a page frame.
5. generic block layer粘合了所有上层和下层组件，因此它有sector，block，segment和pages of data的信息。

即使有不同大小的chunk，它们通常共享相同的物理RAM。如图所示，上层内核组件看见页面是由4个block buffers组成。最后3个blocks被块设备驱动传输，那么他们被插入一个segment。磁盘控制器认为这个segment由6个sector组成。



14.1.1. Sectors

为了可接受的性能，磁盘一次会传输许多相邻的字节。每个块设备的数据传输操作表现为一组相邻的字节，被称为一个sector。磁盘控制器接收控制命令，控制命令把磁盘看成一个大的sector数组。

绝大多数磁盘设备中,sector大小是512字节。注意到,sector是数据传输的基本单元;不可能传输少于一个sector的数据。

Linux中， sector的大小常被设置为512字节;如果一个块设备使用更大的sector,相应的低层块设备驱动将做必要的转换。

14.1.2. Blocks

Block是VFS和文件系统数据传输的基本单元。每个读或写操作作用在块设备文件上是一个"raw"访问,它绕过了基于磁盘的文件系统;内核使用最大块大小(4096字节)来执行它。

每个块有自己的block buffer,它是一个RAM内存区域,被内核用来保存block的内容。当内核从磁盘上读一个block,它将用硬件设备上的数据填充相应的block buffer。同样,当内核在磁盘上写一个block,它将用对应block buffer中的数据更新磁盘。Block buffer的大小总是与对应block大小相等。

每个buffer有一个类型为buffer_head的"biffer head"描述符。这个描述符包含了内核所需的,如何处理这个buffer的所有信息。因而,在操作每个buffer前,内核检查它的buufer head。

14.1.3. Segments

每次I/O操作都会在RAM和磁盘之间传输一些连续的sectors。早期的磁盘控制器只支持"simple"DMA操作:每次操作,数据只在磁盘和物理上连续的RAM之间传输。近期的磁

盘控制器，支持scatter-gather DMA传输：每次操作时，数据能在磁盘和一些不连续的内存区域之间传输。

对每次scatter-gather DMA传输，块设备驱动必须发送给磁盘控制器：
1． 初始的磁盘sector号和要传输的总sectors数目。
2． 一个内存区域描述符链表，每个描述符包括一个地址和一个长度。

磁盘控制器参与了整个数据传输：例如，在读操作中，控制器从邻近的磁盘sectors中取数据然后分散到不同的内存区域中。

为了利用scatter-gather DMA操作，块设备驱动必须以swgment为单位处理数据。一个segment是一个内存页面或内存页面的一部分，它包含了一些邻近的磁盘sectors。因而，一次scatter-gather DMA操作一次使用了许多segments。

如果相应的page frames正好在RAM中连续并且相应的磁盘数据chunks在磁盘上邻近，generic block layer能合并不同的segments。由合并操作形成的内存区域被称为physical segmanet。X86没有IO-MMU，因此hardware segment和physical segment是重合的。

14.2. The Generic Block Layer

generic block layer是一个内核组件，它处理系统中所有块设备的请求，使用该组件后，内核能轻松的完成下列功能：
1． 高地址空间的page frame仅在CPU要访问数据时才映射到内核线性地址空间里，之后会马上解除映射。
2． 实现了零拷贝策略。本质上是把内核中用做I/O传输的buffer的page frame映射到用户线性地址空间。
3． 管理逻辑卷：许多磁盘分区，甚至是在不同块设备上的，能被看为一个单一的分区。
4． 利用了最新的磁盘控制器的高级特性，例如大板载磁盘cache，增强的DMA能力，板载I/O传输请求调度，和其它。

14.2.1. The Bio Structure

generic block layer的核心数据结构是一个descriptor of an ongoing I/O block device operation，被称为bio。每个bio本质上包含了一个磁盘存储区域的标识符，由初始sector号和磁盘存储区的sectors数目组成，还包含了一个或多个segments，描述了使用的内存区域。

每个segment在bio中被一个bio_vec数据结构代表。bio->bi_io_vec指向bio_vec数据结构数组的第一个元素，同时bio->bi_vcnt存储了数组中当前元素的个数。

14.2.2. Representing Disks and Disk Partitions

一个disk是一个被generic block layer处理的逻辑块设备。通常一个disk等同于一个硬件块设备，例如：一个硬盘，一个软盘，或一个CD-ROM。而且，一个disk还能是一个构建在许多物理磁盘分区上的虚拟设备，或RAM中的一些指定页面组成的存储区域。上层的内核组件用相同的方法来操作所有的disk。

一个disk被gendisk对象代表。Flsgs成员储存了关于disk的信息。最重要的flag是GENHD_FL_UP：如果设置了，disk被初始化而且正在工作中。另一个flag是GENHD_FL_REMOVABLE，如果disk支持移除操作，它将被设置。

gendisk->fops指向block_device_operations表,储存了一些对块设备关键操作的专用方法。

硬盘通常被分割为逻辑分区。每个块设备文件可能代表了一个整体disk或其中一个分区。如果一个disk被分割成多个分区,它们的布局保存在hd_struct结构数组中,这个数组地址在gendisk->part中。

当内核在系统中发现一个新的disk,它将调用alloc_disk()函数,分配和初始化一个新的gendisk对象，并且，如果新disk被划分为多个分区,一个合适的hd_struct数组被创建。接着,调用add_disk()函数把新分配的gendisk描述符插入到generic block layer中。

14.2.3. Submitting a Request

让我们来描述当内核提交一个I/O操作请求给generic block layer时,其执行的步骤。我们假设请求的数据chunks在disk上邻近并且内核已经确定好了他们的物理位置。

第一步是执行bio_alloc()函数来分配一个新的bio描述符。接着内核初始化了该bio描述符:

1. bio->bi_sector被设置为数据的初始sector。(如果块设备被分为许多分区,sector号是相对于分区的开始)。
2. bio->bi_size被设置为覆盖请求数据的sector数量。
3. bio->bdev被设置为块设备描述符的地址。
4. bio->bi_io_vec被设置为bio_vec数据结构数组的地址，数组中每个元素描述了一个I/O操作的segment;而且,bi_vcnt成员被设置为bio中segment的总数量。
5. bio->bi_rw被设置为请求的操作的flags。最重要的flag指定了数据传输的方向:READ(0)或WRITE(1)。
6. bio->bi_end_io is set to the address of a completion procedure that is executed whenever the I/O operation on the bio is completed.

一旦bio描述符被恰当的初始化,内核调用generic_make_request()函数,它是generic block layer的主要入口点。函数本质上执行下列步骤:

1. 检查bio->bi_sector没有超过块设备的sectors总数。如果超过了,函数设置bio->bi_flag为BIO_EOF标志,打印一个内核错误消息,调用bio_endio()函数,然后结束。bio_endio()更新bio描述符的bi_size和bi_sector成员,然后调用bio->bi_end_io方法。bio->bi_end_io方法的实现本质上依赖于触发I/O数据传输的内核组件;
2. 得到与块设备相关的请求队列q;它的地址在bio->bi_bdev指向的块设备描述符的bd_disk成员中。
3. 调用block_wait_queue_remap()来检查当前正在使用的I/O scheduler是否被动态的替换了;如果是,函数使进程进入睡眠,直到新的I/O scheduler开始。
4. 调用blk_partition_remap()来检测块设备是否对应的是一个disk分区。这种情况下,函数从bio->bi_bdev中得到分区的hd_struct描述符，然后执行下列步骤:
   a. 根据数据传输的方向,更新hd_struct描述符的read_sectors和reads成员,或者write_sectors和writea成员。
   b. 调整bio->bi_sector,把它的相对于分区起始位置的sector号转换为相对于整个磁盘的sevtor号。
   c. 设置bio->bi_bdev为指向整个disk的块设备描述符。

从现在起，generic block layer,I/O scheduler,和设备驱动忘记了磁盘分区，而将直接操作整个disk。

5. 调用q->make_request_fn方法把bio请求插入请求队列q。
6. 函数返回。

14.3. The I/O Scheduler

虽然块设备驱动能够一次传输一个单独的sector,但block I/O layer并没有为一个单独的sector执行一次I/O操作;因为这将导致糟糕的磁盘性能。内核试图尽可能的把许多sectors聚集在一起,然后把它们作为一个整体来处理,因而减少了平均磁头移动次数。

当一个内核组件打算读取或写一些disk数据,它会创建一个块设备请求。这个请求本质上描述了请求的sectors和执行的操作类型。而且，一个请求并不是在刚被创建就会被内核满足,I/O操作会被调度然后会过一段时间才被执行。这种人工延时对提升块设备的性能是很关键的机制。当一个新的块数据被请求时,内核检测这个请求是否能通过稍稍扩大还在等待中的之前的请求来满足它。因为disks倾向于顺序的被访问,这个简单的策略是十分有效的。

延迟I/O操作请求使块设备处理复杂化。块设备驱动自己不能被阻塞,否则其它试图访问驱动所控制的disk时也会被阻塞。

为了使块设备驱动不被挂起,每个I/O操作被异步的处理。块设备驱动是中断驱动的:generic block layer调用I/O scheduler来创建一个新的块设备请求或者扩大已经存在的请求,然后结束。过了段时间,块设备驱动被激活,调用strategy routine来选择一个pending请求,然后发送合适的命令给disk controller来满足该请求。当I/O操作结束时,disk控制器产生一个中断,然后如果有必要,相应的处理程序再次调用strategy routine来处理另一个panding请求。

每个块设备驱动维护一个自己的请求队列,它包含了一个pending请求链表。如果disk控制器正在处理多个disks,那么对每个物理块设备通常有一个请求队列。I/O scheduler在每个请求队列上分别执行,这样就增加了disk性能。

14.3.1. Request Queue Descriptors

每个请求队列由数据结构request_queue表示。请求队列中每个元素是request descriptor(它是request数据结构)； request_queue->queue_head是请求队列的链表头，request descriptor通过request->queuelist挂入这个对列头。请求队列中元素的顺序对每个块设备驱动是确定的;I/O scheduler提供了许多预先定义的排序这些元素的方法。

request_queue->backing_dev_info是一个backing_dev_info类型的小对象,它保存了关于硬件块设备的I/O数据流的信息。例如:它保存了关于read-ahead信息和关于请求队列的拥塞状态信息。

14.3.2. Request Descriptors

每个块设备的pending请求被一个request descriptor代表,它的数据结构是request。

每个请求由一个或多个bio结构组成。初始时,generic block layer创建一个仅包含一个bio的请求,I/O scheduler可能通过加入一个新的segmente到bio中或链入一个

新的bio来"extend"这个请求。这种情况发生在新数据与bio请求中的数据在物理上邻近。request->bio指向请求中的第一个bio,同时request->biotail指向最后一个bio。rq_for_each_bio宏实现了一个循环来迭代所有request对象中的bio。

request descriptor许多成员可能会动态的改变。例如,一旦bio结构引用的数据传输完毕,bio成员就更新为下一个bio。同时,新bio能被加入request descriptor中bio链表的尾部,因此biotail成员也要更新。

14.3.2.1. Managing the allocation of request descriptors

在disk操作活动过多时,空闲动态内存总量的限制将成为一个进程添加新请求到一个请求队列q中的瓶颈。为了应对这种情况,每个request_queue描述符包含了一个request_list数据结构,它包含了:

1. 一个指向请求描述符的memory pool的指针。
2. 2个计数器,保存了分配给READ和WRITE请求的request descriptor。
3. 2个flags指明了最近分配给READ或WRITE的请求是否失败了。
4. 2个等待队列保存了睡眠进程,它们分别正在等待READ和WRITE request descriptor可用。
5. 一个等待队列,其中的进程正在等待一个请求队列被刷新。

blk_get_request()函数尝试从一个给定的请求队列的memory pool中得到一个空闲的request descriptor;如果内存很稀缺并且memory pool被耗尽了,函数要么使当前进程睡眠要么返回NULL。如果分配成功了,函数把请求队列的request_list数据结构的地址存放在request descriptor的rl成员中。blk_put_request()用来释放一个request descriptor。

14.3.2.2. Avoiding request queue congestion

每个请求队列有一个允许pending的请求最大数量。request queue descriptor的nr_requests成员保存了每个传输方向上最大允许的pending请求数目。缺省的,一个队列最多有128个读请求和128个写请求。如果pending读(写)请求数量超过了nr_requests,队列设置request queue descriptor的queue_flags成员为QUEUE_FLAG_READFULL(QUEUE_FLAG_WRITEFULL)标记,表示队列已满,在那个方向上正在尝试添加请求的可阻塞的进程被挂入requset_list中的对应睡眠队列。

一个填满了的请求队列对系统的性能有消极的冲击，因为它强迫许多在等待I/O数据传输完成的进程进入睡眠。那么,如果给定方向上的pending requests数目超过了请求队列描述符中nr_congestion_on值,内核认为队列进入拥塞了并且试图放慢新请求的创建速度。当pending requests数目低于nr_congestion_off值,拥塞了的队列变成非拥塞。blk_congestion_wait()函数使当前进程进入睡眠,直到请求队列变成为非拥塞或超时。

14.3.3. Activating the Block Device Driver

把块设备驱动的激活推迟是有利的,可以增加把I/O请求聚集成邻近磁盘块的机会。这种推迟技术被称为device plugging and unplugging。即使有请求在设备的队列中,设备驱动也不会激活。

blk_plug_device()函数plug一个请求队列,这个请求队列被块设备驱动服务。本质上,函数接收一个请求队列描述符地址q作为参数。它设置q->queue_flags的QUEUE_FLAG_PLUGGED位;接着,他重启内嵌在q->unplug_timer中的动态定时器。

blk_remove_plug()函数unplug一个请求队列q:它清除QUEUE_FLAG_PLUGGED flag 并且退出q->unplug_timer动态定时器的执行。当所有可以合并的请求被加入队列后,这个函数能被内核显式的调用。而且,如果队列中的pending request数目超过请求队列描述符中的unplug_thres的值,I/O scheduler unplug一个请求队列。

如果一个设备在时间间隔q->unplug_delay后还是plugged状态,动态定时器超时,那么blk_unplug_timeout()函数执行。kblockd内核线程被唤醒,它会服务kblockd_workqueue工作队列。这个内核线程执行地址保存在q->unplug_work中的函数:blk_unplug_work()。接着,这个函数调用q->unplug_fn方法,这个方法通常是指向函数:generic_unplug_device()。generic_unplug_device()函数执行unplug块设备:首先,它检查队列是否仍然是活跃的;然后,它调用blk_remove_plug();最后,它执行了strategy routine:request_fn方法来开始处理队列中的下一个请求。

14.3.4. I/O Scheduling Algorithms

当一个新的请求被加入到一个请求队列,generic block layer调用I/O scheduler来决定新请求在队列中的准确位置。I/O scheduler试图保持请求队列按照sctor位置排序。如果是按链表中顺序来处理请求,那么disk寻到总数将明显减少,因为磁头以线性方式从内往外移动。

在高负载情况下,严格按照sctor号排序的I/O scheduler算法将工作的不好。在这种情况下,数据传输完成时间依赖于数据在磁盘上的物理位置。那么,如果一个设备驱动正在处理靠近队列顶部的请求,而且新请求都是些sector号较小的,那么队列尾部的请求将很容易饥饿。

当前,Linux2.6提供了4种不同类型的I/O scheduler elevators:”Anticipatory”,”Deadline”,”CFQ(Complete Fairness Queueing)”,和”Noop(No Operation)”。缺省的是使用”Anticipatory”。

用在一个请求队列中的I/O scheduler算法用一个elevator对象来代表,其类型为elevator_t;它的地址被储存在请求队列描述符的elevator成员中。elevator对象包含了许多方法,它们覆盖了elevator的所有可能的操作。elevator对象也储存了一个表的地址,这个表包含了所有的需要用来处理请求队列的信息。而且,每个请求队列描述符包含了elevator_private成员,它指向一个额外的数据结构,I/O scheduler将用它来处理请求。
通常来说,所有算法利用了一个dispatch queue,它包含了所有请求,这些请求根据被设备驱动处理的顺序来进行排序。被设备驱动服务的下一个请求总是dispatch queue的第一个元素。dispatch queue是真正的请求队列,它以请求队列描述符的queue_head为队列头。几乎所有的算法利用了额外的队列来区分和排序请求。所有算法允许设备驱动添加bios到已存在的请求中,以及如果有必要就合并2个邻近的请求。

14.3.4.1. The "Noop" elevator

没有排序队列:新请求总是要么添加到dispatch queue前面要么添加到尾部,并且下一个要被处理的请求总是队列中第一个。

14.3.4.2. The "CFQ" elevator

"Complete Fairness Queueing" elevator的主要目标是确保disk I/O带宽在所有触发了I/O请求的进程中公平的分配。elevator利用了大量缺省的已排序队列,缺省64个。保存了来自不同进程的请求。无论何时,一个请求被提交给elevator,内核调用一个hash函数把当前进程的线程组标识符转换为队列索引;那么elevator把新请求插入索引对应的队列的尾部。所以,来自相同进程的请求被挂入相同的队列。

为了填充dispatch queue,elevator本质上以round-robin方式搜索了I/O输入队列,选择第一个非空队列,并且把a batch of requests从那个队列中移动到dispatch queue尾部。

14.3.4.3. The "Deadline" elevator

加上dispatch queue,”Deadline” elevator一共利用了4个队列。它们中的两个已排序队列分别包含了读和写请求,这两个队列根据他们的初始sector号进行的排序。另外两个deadline queues包含了相同的读和写请求,不过是根据它们的”deadlines”来进行排序。这些队列用来避免某些请求的饥饿情况发生。一个请求的deadline本质上是一个定时器,它从请求被传给elevator开始计时。deadline确保了当一个请求等待了一个很长时间后,能被scheduler注意到。

14.3.4.4. The "Anticipatory" elevator

略

14.3.5. Issuing a Request to the I/O Scheduler

generic_make_request()函数调用请求队列描述符的make_request_fn方法来传输一个请求到I/O scheduler。这个方法通常被__make_request()函数实现;它接收参数:一个request_queue描述符q和一个bio描述符bio,并执行下列操作:

1. 如果有需要就调用k_queue_bounce()函数来设置一个bounce buffer。如果bounce buffer被创建了, __make_request()函数就在它之上进行操作,而不在原来的bio上。
2. 调用I/O scheduler函数elv_queue_empty()来检查请求队列中是否有pending请求。注意到,dispatch queue可能是空的,但是I/O scheduler的其它队列可能包含了pending请求。如果没有pending请求,它就调用blk_plug_device()函数并跳转到步骤5。
3. 这一步里,请求队列中包含了pending请求。调用elv_merge()来检查新的bio是否能被合并进一个已存在的请求中。函数可能返回3个可能的值:

- ELEVATOR_NO_MERGE:bio不能被加进一个已存在的请求中,函数跳转到5。
- ELEVATOR_BACK_MERGE: bio可以被添加到一些struct request **req中的最后:在这种情况下,函数调用q->back_merge_fn方法来检查,请求是否能被扩展。如果不能,函数跳转到5。否则它把bio描述符插入到req的链表的尾部并且更新req的成员。然后,它尝试合并下一个请求。(新插入的bio可能正好填充了2个请求之间的hole)。
- ELEVATOR_FRONT_MERGE:bio可以被添加到一些struct request **reqreq中的开始处:在这种情况下,函数调用q->front_merge_fn方法来检查,请求是否能被扩展。如果不能,函数跳转到5。否则它把bio描述符插入到req的链表的首部并且更新req的成员。然后,它尝试合并前一个请求。(新插入的bio可能正好填充了2个请求之间的hole)。

4. bio已经被合并进一个已经存在的请求中。跳转到7并结束函数。

5. 这里,bio必须插入一个新的请求中。分配一个新请求描述符。如果没有空闲内存,函数挂起当前进程，除非bio->bi_rw的BIO_RW_AHEAD flag被设置了,它意味着I/O操作是一个read-ahead;在这种情况下,函数调用bio_endio()并结束:数据传输并没有执行。
6. 初始化请求描述符的成员。
7. 各种工作都完成了。在结束之前,它检查bio->bi_rw的BIO_RW_SYNC flag是否被设置。如果设置了，调用generic_unplug_device()来unplug驱动。
8. 结束。

14.4. Block Device Drivers

块设备驱动是Linux block subsystem的最低一级组件。它从I/O scheduler取得请求然后处理它。

块设备驱动被整合进了device driver model中。因此，驱动引用了一个device_driver描述符;而且,没disk处理都与一个device描述符联系。这些描述符都是通用的,因此block I/O subsystem必须在系统中为每个块设备存储额外的信息。

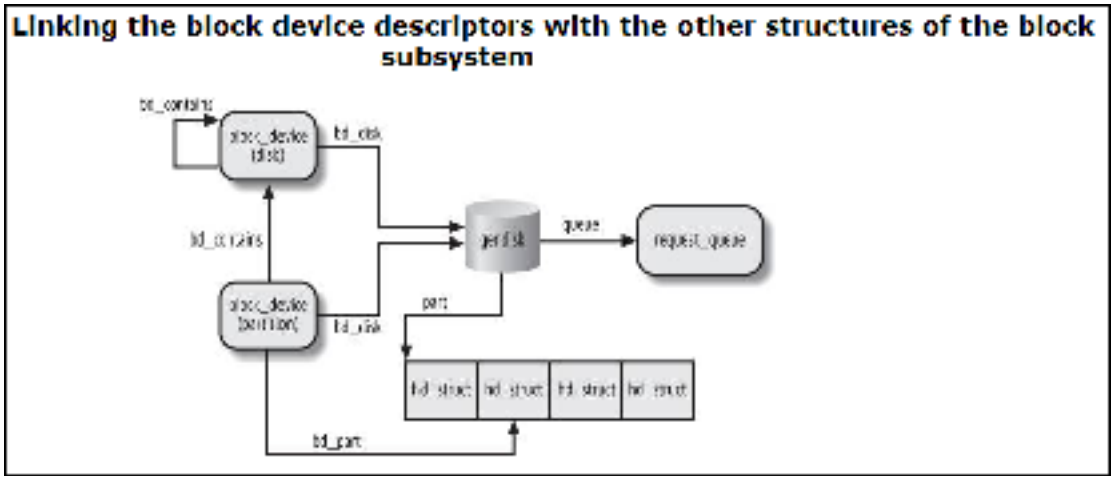
14.4.1. Block Devices

一个块设备驱动可以处理多个块设备。每个块设备用block_device描述符代表。

所有的块设备描述符被插入一个全局链表,链表头保存在all_bdevs中;块设备描述符通过bd_list链入该链表。

如果块设备描述符引用的是一个磁盘分区， block_device->bd_contains成员指向整个块设备的描述符,同时， block_device->bd_part指向hd_struct描述符。如果块设备描述符引用的是整个磁盘， block_device->bd_contains成员指向块设备自己，并且block_device->bd_part_count记录了磁盘上的分区被打开几次。

block_device->bd_holder保存了一个代表holder of the block device的线性地址。这个holder不是块设备驱动而是一个内核组件,它使用设备并且has exclusive, special privileges ((例如,它能自由的使用块设备描述符的bd_private成员)。典型的,一个块设备的holder是安装在其上的文件系统。另一种常用情况是,当一个块设备文件被独占的打开,holder就是对应的file对象。



14.4.1.1. Accessing a block device

当内核接收到一个打开块设备文件的请求,它必须首先确定设备文件是否已经打开。事实上,如果文件已经打开了,内核不须要创建和初始化一个新的块设备描述符;而是更新已经存在的块设备描述符。可能块设备文件有相同的major和minor号但是有不同的路径名,但实际上它们引用的是同一个块设备。所以,内核不能通过块设备文件的inode对象是否存在来决定它是否已经在使用中。

major和minor号之间的联系以及对应的块设备描述符通过bdev特殊文件系统维护。每个块设备描述符的bd_inode指向对应的bdev inode;这个inode编码了块设备的major,minor号以及对应描述符的地址。

bdget()函数接收块设备的major,minor号作为参数:它查询bdev文件系统中对应的inode;如果inode不存在,函数分配一个新的inode和新的块设备描述符。否则,函数返回对应块设备的块设备描述符地址。

一旦块设备描述符已经被找到,内核能通过检查bd_openers成员来确定块设备当前是否在使用中。内核也维护了一个关于打开的块设备文件的inode对象链表。inode对象通过i_devices成员链入块设备描述符的bd_inodes。

14.4.2. Device Driver Registration and Initialization

14.4.2.1. Defining a custom driver descriptor

首先,设备驱动需要一个专用的foo_dev_t类型的描述符foo,它包含了用于驱动硬件设备的数据。对每一个设备,描述符储存了这些信息: I/O ports,IRQ线,设备的内部状态等等。描述符必须也包含一些被block I/O subsystem使用的成员:

```
struct foo_dev_t {
    [...]
    spinlock_t lock;
    struct gendisk *gd;
    [...]
} foo;
```

该章以下略，以后再说。

14.5. Opening a Block Device File

内核每当在磁盘或分区上加载一个文件系统时,或当一个swap分区被激活时以及当用户进程在块设备文件上使用open()系统调用时,内核都会打开块设备文件。无论哪种情况,内核本质上是执行相同的操作:它查找块设备描述符,为即将到来的数据传输建立文件操作的方法。

在VFS层,dentry_open()函数使已打开文件的file对象的方法关联到特定函数。在我们这里,file->f_op被设置成def_blk_fops表的地址。

这里,file->open被关联到blkdev_open()函数,它接收参数inode和filp,执行下列步骤:

1. 执行bd_acquire(inode)来得到块设备描述符的地址,存入局部变量bdev中。这个函数执行下列步骤:

a. 检查inode->i_bdev是否为NULL;如果为非NULL,说明块设备文件已经被打开了,inode->i_bdev保存了对应块设备描述符的地址。这种情况下,函数递增inode->i_bdev->bd_inode的值,并且返回inode->i_bdev指向的描述符。

b. 这里,块设备文件还没有被打开。执行bdget(inode->i_rdev)得到对应块设备描述符的地址。如果描述符不存在,bdget()分配一个;注意到,描述符可能已经存在了,例如该块设备被另一个块设备文件访问了。
c. 保存块设备描述符地址到inode->ibdev。
d. 设置inode->i_mapping为bdev inode中对应值。这个指针指向address space object。
e. 把inode插入到块设备描述符的已打开inodes链表中,链表头为bdev->bd_inodes。
f. 返回bdev。
2. 设置filp->i_mapping为inode->i_mapping。
3. 得到与这个块设备相关的gendisk描述符地址:

```
disk = get_gendisk(bdev->bd_dev, &part);
```

如果块设备是一个磁盘分区, get_gendisk()函数返回了分区的index,保存在part变量中;否则,part被设为0。

4. 如果bdev->bd_openers不等于0,块设备已经被打开了。检查bdev->bd_containers:

a. If it is equal to bdev, the block device is a whole disk: invokes the bdev->bd_disk->fops->open block device method, if defined, then checks the bdev->bd_invalidated field and invokes, if necessary, the rescan_partitions( ) functions (see comments on steps 6a and 6c later).
b. If it not equal to bdev, the block device is a partition: increases the bdev->bd_contains->bd_part_count counter.

Then, jumps to step 8.

5. Here the block device is being accessed for the first time. Initializes bdev->bd_disk with the address disk of the gendisk descriptor.
6. If the block device is a whole disk (part is zero), it executes the following substeps:

a. If defined, it executes the disk->fops->open block device method: it is a custom function defined by the block device driver to perform any specific last-minute initialization.
b. Gets from the hardsect_size field of the disk->queue request queue the sector size in bytes, and uses this value to set properly the bdev->bd_block_size and bdev->bd_inode->i_blkbits fields. Sets also the bdev->bd_inode->i_size field with the size of the disk computed from disk->capacity.
c. If the bdev->bd_invalidated flag is set, it invokes rescan_partitions( ) to scan the partition table and update the partition descriptors. The flag is set by the check_disk_change block device method, which applies only to removable devices.

7. Otherwise if the block device is a partition (part is not zero), it executes the following substeps:

a. Invokes bdget( ) againthis time passing the disk->first_minor minor numberto get the address whole of the block descriptor for the whole disk.
b. Repeats steps from 3 to 6 for the block device descriptor of the whole disk, thus initializing it if necessary.
c. Sets bdev->bd_contains to the address of the descriptor of the whole disk.
d. Increases whole->bd_part_count to account for the new open operation on the partition of the disk.
e. Sets bdev->bd_part with the value in disk->part[part-1]; it is the address of the hd_struct descriptor of the partition. Also,

executes kobject_get(&bdev->bd_part->kobj) to increase the reference counter of the partition.
f. As in step 6b, sets the inode fields that specify size and sector size of the partition.
8. Increases the bdev->bd_openers counter.
9. If the block device file is being opened in exclusive mode (O_EXCL flag in filp->f_flags set), it invokes bd_claim(bdev, filp) to set the holder of the block device (see the section "Block Devices" earlier in this chapter). In case of errorblock device has already an holderit releases the block device descriptor and returns the error code -EBUSY.
10. Terminates by returning 0 (success).

Chapter 15. The Page Cache

15.1. The Page Cache

page cache是Linux内核使用的最主要的一种disk cache。在绝大多数情况下,内核在读或写磁盘时，引用的都是page cache。用户进程发出读磁盘请求时,如果页面不在cache中,内核从磁盘读取数据,创建一个新的entry。如果有足够的空闲内存,页面将在cache中一直存在,它能被不同进程重复使用而不用访问磁盘。相似的,用户进程发出写磁盘请求时,如果对应页面不在cache中,创建一个新的entry,用要写入的数据填充它。写操作的I/O数据传输并不是立即执行:disk更新被推迟了几秒,那么就给进程一个修改要写的数据的机会。(换句话说,内核实现了deferred write operations)。

在page cache中的页面有如下类型:
1. 包含常规文件的页面。
2. 包含目录的页面。
3. 包含从块设备文件直接读取的数据。
4. 包含用户进程的被交换到磁盘上的数据。
5. 属于特殊文件系统的页面,例如用于进程间通信的shm文件系统。

每个page cache中的页面包含的数据属于某个file。这个file或精确的说是file的inode被称为页面的owner。

实际中,read()和write()文件操作依赖于page cache。唯一的例外发生在当一个进程以O_DIRECT flag,打开一个文件:在这种情况下,page cache被绕过了,I/O传输利用了用户进程空间的buffers;一些数据库应用利用了O_DIRECT flag,它们就能使用自己的disk caching algorithm。

内核设计者实现page cache是为了满足2个主要的需求:
1. 快速搜索一个页面,这个页面包含了与给定owner相关的数据。为了最大限度的利用page cache,搜索应该是一个很快的操作。
2. 跟踪cache中的每个页面在读写其内容时的方式。例如, reading a page from a regular file, a block device file, or a swap area must be performed in different ways,内核必须根据页面的owner来选择合适的操作。

保存在page cache中的信息以一个整页面为单位。一个page cache并不需要包含物理上连续的磁盘块,因此它不能用设备号加块号来标识。Instead, a page in the page cache is identified by an owner and by an index within the owner's data usually, an inode and an offset inside the corresponding file.

15.1.1. The address_space Object

page cache的核心数据结构是address_space对象,这个数据结构内嵌在拥有该page的inode对象中(有一个例外,被交换出去的页面有一个通用的address_space对象,它没有包含在任何inode中)。许多cache中的页面可能引用到同一个owner,因而它们被链接到同样的address_space对象。这个对象在owner的页面和一组对页面的操作之间建立了链接。

每个页面描述符有两个成员:mapping和index。mapping指向拥有该page的inode的address_space对象。index表明了page的数据在owner的磁盘镜像中的位置。当在page cache中搜寻一个页面时将用到这两个成员。

令人吃惊的是,page cache可能会包含同一个磁盘数据的多份副本。例如,同一个4K的磁盘数据被用如下方式访问:

1. 读文件;数据被包含在owner为常规文件的inode的page中。
2. 从设备文件中读取块;那么,数据被包含在owner为the master inode of the block device 的页面中。

于是,同一个磁盘块出现在2个引用不同address_space对象的不同页面中。

如果一个page cache中的页面的owner是一个file,那么address_space对象内嵌在VFS inode对象的i_data成员中。inode->i_mapping总是指向对应的address_space对象。address_space->host指向address_space内嵌在的inode对象。

如果页面所属的文件是存储在EXT3文件系统中,那么,page的owner是文件的inode并且对应的address_space对象存储在VFS的inode对象的i_data成员中。inode->i_mapping指向同一个inode->i_data成员,对应address_space->host指向这个inode。

有时,情况会变得更复杂。如果页面包含的数据是从块设备文件中读取的,那么address_space对象将内嵌在"master" inode of the file in the bdev special filesystem associated with the block device.所以,块设备文件的inode->i_mapping指向内嵌在master inode中的address_space对象。从这个块设备文件中读取的数据,在page cache中的页面指向相同的address_space对象,即使它们是从不同的块设备文件(但指向的是同一个块设备)中访问的。

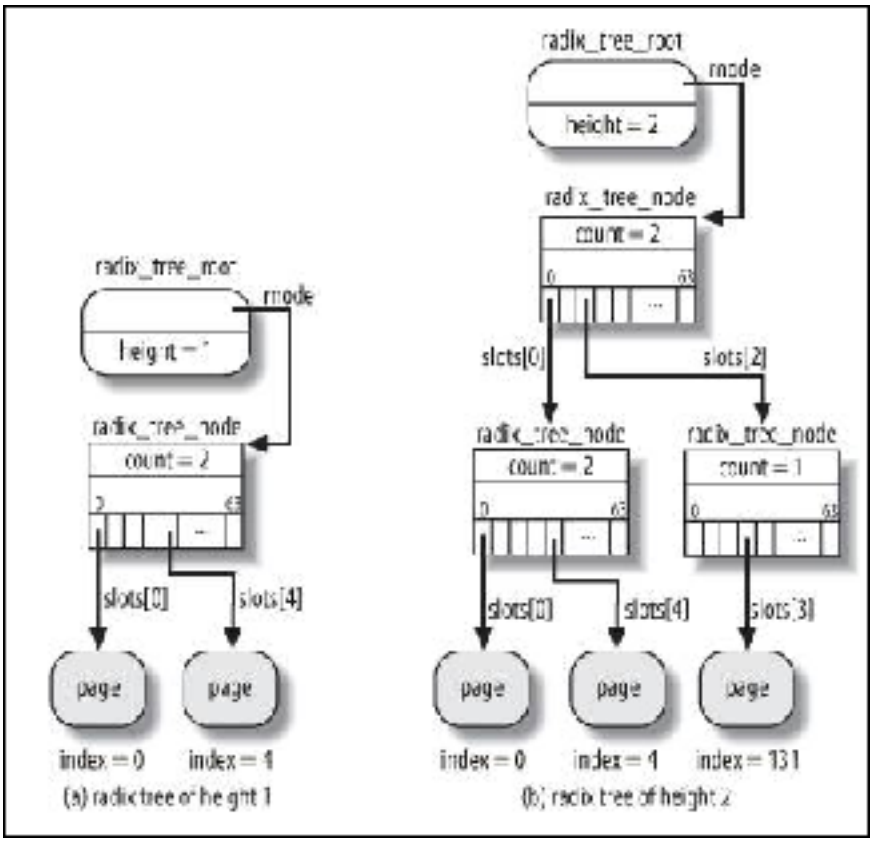
address_space->a_ops是最重要的成员。它指明了如何处理owner的页面。

15.1.2. The Radix Tree

Linux中,文件可能很大,因此当访问一个很大的文件时,page cache可能被填充了过多的页面,以至于顺序访问它们是及其耗时的。为了使page cache查询效率更高,Linux2.6利用了一个搜索树的集合,每个搜索树用于一个address_space对象。

address_space->page_tree对象是一棵radix tree的根。给定一个page index，它代表了在owner的磁盘镜像中该page的位置，内核能执行一个快速查询操作来确定请求的页面是否在page cache中。

radix tree的每一个node有最多64个指针指向其它node或指向page描述符。在底部的nodes(叶子节点)保存了指向page描述符的指针，而内节点保存的是指向其它node的指针。每个node的数据结构类型是:radix_tree_node，它包含了3个成员:slots是一个含64个指针的数组，count是一个计数器，表明了有多少个指针为非NULL，tags是一个二维数组。每棵radix tree的树根数据结构类型是:radix_tree_root，有3个成员:height代表了当前树的高度(除了叶子的层数)，gfp_mask指明了分配新node的时的flags，rnode指向树第一层node。



因为32-bit体系下，page index最多32位，所以radix tree最多6层。radix tree中查找方式类似于地址映射，不过， page的index中用来定位的部分依赖于radix tree的高度。如果radix tree高度为1，它只能代表0～63，那么page的index的低6位被解释成第一层中唯一一个node的slots数组的下标。如果radix tree高度为2，那么树能代表0~4095;page的index的低12位用来定位，这12位被分割成2部分，每个部分6 bit;高6 bit作为第一层node的索引，第6 bit作为第二层node的索引。如果高度为6，则最高2位用于第一层node，接下来6位用于第二层node，如此下去。

如果radix tree中的最高的索引比需要加入页面的索引小，那么内核递增了树高度。

15.1.3. Page Cache Handling Functions

15.1.3.1. Finding a page

find_get_page()函数接收2个参数:指向address_space对象的指针，一个offset。它请求了address_space的spin lock和调用radix_tree_lookup()函数来搜索

radix_tree中对应offset的叶子node。这个函数,从树的根依次向下访问。如果碰见一个NULL指针,函数返回NULL;否则,返回叶子node的地址,也就是page描述符的地址。如果请求的页面找到了, find_get_page()递增其使用计数,释放spin lock并且返回地址;否则,函数释放spin lock和返回NULL。

find_get_pages()函数类似于find_get_page()函数,但是它查找的是一组index连续的pages。函数接收3个参数: 指向address_space对象的指针,一个offset,需要查找的页面数,一个数组指针用来保存页面描述符。

15.1.3.2. Adding a page

add_to_page_cache()函数把一个page描述符插入page cache。它接收的参数有:page--页面描述符地址,mapping--address_space对象地址,offset--address space中的page index,gfp_mask。函数执行下列步骤:

1. 调用radix_tree_preload(),它关闭了内核抢占并且用一个新分配的radix_tree_node结构填充per-CPU变量radix_tree_preloads。分配新的radix_tree_node结构是通过radix_tree_node_cachep slab分配器来分配。如果radix_tree_preload()分配失败, add_to_page_cache()函数将终止并返回错误码-ENOMEM。否则,如果radix_tree_preload()分配成功了,add_to_page_cache()能被确保即使在空闲内存不足时,也能成功插入新的页面描述符。
2. 请求mapping->tree_lock spin lock,注意到,之前内核已经关闭抢占了。
3. 调用radix_tree_insert()插入新的node到树中。函数执行下列步骤:

a. 调用radix_tree_maxindex()得到在当前高度下,能插入radix tree的最大索引。如果新页面索引大于刚得到的最大索引,则调用radix_tree_extend()通过添加适当数目的node来增加树高。新的nodes通过执行radix_tree_node_alloc()函数来分配,它试图从slab分配器中分配,或从radix_tree_preload中分配。
b. 从mapping->page_tree开始遍历树,直到叶节点。
c. 把page描述符地址保存在最后访问到的node的合适的slot中,并且返回0。

4. 递增page->_count使用计数。
5. 因为页面是新的,它的内容是非法的:函数设置page frame的PG_locked flag,防止页面被其它内核控制流同时访问。
6. 初始化page->mapping和page->index。
7. 递增mapping->nrpages。
8. 释放address space的spin lock。
9. 调用radix_tree_preload_ebd()开启内核抢占。
10. 返回0。

15.1.3.3. Removing a page

remove_from_page_cache()函数从page cache中移走一个page描述符。执行步骤为:

1. 请求page->mapping->tree_lock spin lock并且关中断。
2. 调用radix_tree_delete()从树中删除node。函数接收参数:树根地址,要删除的页面索引。执行下列步骤:

a. 从根开始遍历,直到寻找到对应的叶node。在遍历的同时,构建一个radix_tree_path结构数组,描述了从根到对应叶子的遍历过得组件信息。

b. 在radix_tree_path结构数组中,从最后一个node(包含了page描述符)开始循环。对每个node,设置对应slots为NULL并递减count成员。如果count变成0,从树中移走node并释放radix_tree_node结构。
c. 返回指向被移走的page描述符指针。
3. 设置page->mapping为NULL。
4. 递减page->mapping->nrpages计数器。
5. 释放page->mapping->tree_lock,开启中断,结束。

15.1.3.4. Updating a page

Read_cache_page()函数确保cache包含了一个最新版本的给定页面。它的参数是:mapping指针指向一个address_space对象,偏移值index指定了请求的页面,filler指针指向一个函数用来从磁盘读取页面数据(通常这个函数实现了addree_space的readpage方法),data指针只想传递给filler的参数。下面是函数执行过程的简单描述:

a. 调用find_get_page()检查页面是否已经在page cache中。
b. 如果页面不再page cache中,它执行下列子过程:
1. 调用alloc_pages()分配一个新page frame。
2. 调用add_to_page_cache()把页面描述符插入page cache。
3. 调用lru_cache_add()把页面插入zone的非活跃LRU链表。
c. 这里,页面已经在page cache中了。调用mark_page_accessed()来记录下该页面已经被访问过了。
d. 如果页面不是最新的,它调用filler函数从磁盘中读取页面。
e. 返回页面描述符地址。

15.1.4. The Tags of the Radix Tree

page cache不仅允许内核快速的找到包含块设备中指定数据的页面,而且允许内核快速的找到处于特定状态的页面。

例如,假设内核必须在cache中找到所有dirty页面。如果通过顺序遍历所有radix tree中的page描述符来收集,那么当只有很少dirty页面时,效率将及其低下。为了能更快的搜索所有dirty页面,每个radix tree中的中间node包含了一个dirty tag;如果该node的孩子node中,至少有一个dirty tag置位,则父node的dirty tag置位。最底层的node的dirty tag通常是从page描述符的PG_dirty flag拷贝而来。使用这种方法后,内核每次寻找dirty页面,它能忽略dirty tag为0的node。

15.2. Storing Blocks in the Page Cache

Linux2.6内核使用了buffer page来作为磁盘block buffer。一个buffer page有一些描述符:buffer heads,它主要用来在page中快速的定位磁盘中block地址。实际中,保存在page cache的页面中的chunks不要求在磁盘上是邻近的。

15.2.1. Block Buffers and Buffer Heads

每个block buffer有一个buffer head描述符,类型为:buffer_head。这个描述符包含了足够的信息,使内核能知道如何来处理block;因此,在操作每个block前,内核会检查它的buffer head。

buffer_head->b_bdev标识了块设备。buffer_head->b_blocknr保存了逻辑块号，也就是block在磁盘或分区中的index。

buffer_head->b_data指明了block buffer在buffer page中的位置。如果页面是在高内存中，b_data包含的是block buffer相对于该page的偏移；否则，b_data包含的是block buffer的线性地址。

buffer_head->b_state存储的是一些flags。

15.2.2. Managing the Buffer Heads

buffer heads有自己的slab分配器，它的kmem_cache_s描述符保存在bh_cachep变量。alloc_buffer_head()和free_buffer_head()函数分别用来得到和释放一个buffer head。

15.2.3. Buffer Pages

当内核必须单独寻址一个block时，它将引用保存了这个block buffer的buffer page，并且检查对应的buffer head。

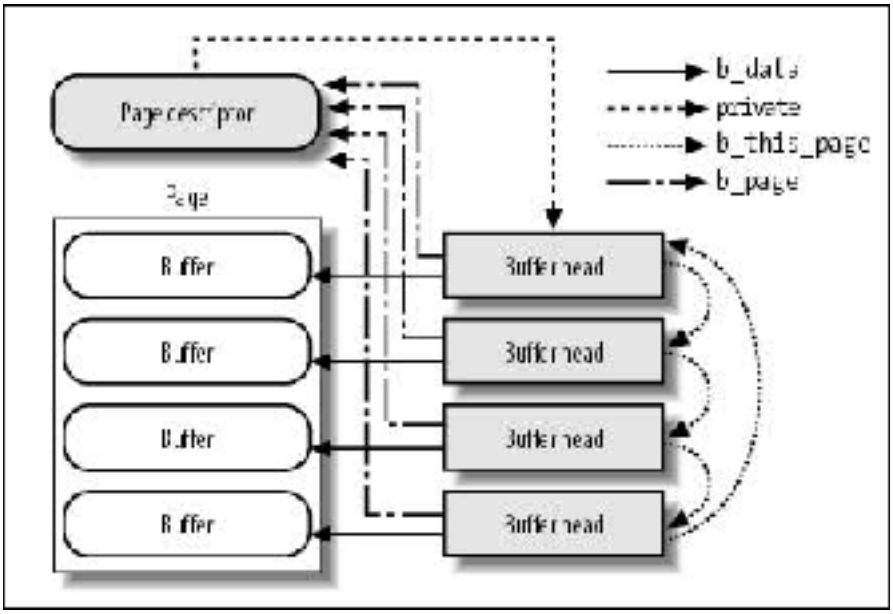
有两种情况内核会创建buffer pages:

- 读或写某个文件的页面，但文件并没有保存在连续的磁盘块。这种情况发生在当文件系统分配非连续的块给文件或文件含有"holes"。
- 访问一个单一的磁盘块(例如，当读取一个超级块或一个inode block)。

在第一种情况下，buffer page的描述符被插入常规文件的radix tree。buffer heads保存了块设备和逻辑块号。

在第二种情况下，buffer page的描述符被插入radix tree，这个radix tree的根在bdev特殊文件系统中块设备的inode的address_space对象。这种buffer pages必须满足一个限制：所有的block buffers必须引用磁盘上邻近的blocks。

在本章中，我们将关注第二种buffer pages，它被称为block device buffer pages。



15.2.4. Allocating Block Device Buffer Pages

当内核发现page cache没有包含需要的页面,则内核分配一个新的block device buffer page。

为了添加一个block device buffer page进page cache,内核调用grow_buffers()函数,它接收3个参数: struct block_device *bdev, sector_t block, int size。

函数本质上是执行了下列动作:

1. 计算请求块所在的页面的index。
2. 如果有需要,调用grow_dev_page()创建一个新的block device buffer page。函数执行下列步骤:

a. 调用find_or_create_page()。
b. 现在请求的页面在page cache中,函数拥有了页面描述符的地址。函数检测PG_private flag;如果是NULL,说明页面还不是buffer page:跳转到2e。
c. 页面已经是一个buffer page。检测page->private->bh->size是否等于请求的块的大小;如果相等,说明在page cache中找到的page是合法的:跳转到2g。
d. 页面的块大小不正确:调用try_to_free_buffers()来释放错误的buffer heads。
e. 调用alloc_page_buffers()为请求的块分配buffer heads,插入b_this_page单链表。而且初始化b_page为page描述符地址,初始化b_data为偏移量或一个线性地址(由是否为高地址内存来决定)。
f. 保存第一个buffer head的地址到page->private,设置PG_private,递增页面使用计数。
g. 调用init_page_buffers()函数来初始化b_bdev,b_blocknr,b_bstate。所有的块在磁盘上是邻近的,那么逻辑块号是连续的并且能轻松的从参数block得到。
h. 返回页面描述符地址。

3. 解锁page。
4. 递减页面的使用计数。
5. 返回1。

15.2.5. Releasing Block Device Buffer Pages

当内核试图得到额外的空闲内存时, block device buffer pages将被释放。如果buffer page含有dirty或locked buffers时,它就不能被释放。内核调用try_to_release_page()函数来释放buffer pages。

15.2.6. Searching Blocks in the Page Cache

当内核需要读或写一个物理设备的某个块,它必须检查所需要的block buffer是否已经在page cache中。Searching the page cache for a given block buffer specified by the address bdev of a block device descriptor and by a logical block number nr is a three stage process:

1. 通过bdev->bd_inode->i_mapping得到address_space对象的指针。
2. 通过bdev->bd_block_size得到设备的块大小,并且计算包含了该block的页面的index。这只需要通过把逻辑块号偏移就可以得到。
3. 在块设备的radix tree中搜索buffer page。在找到page描述符后,内核将访问包含block buffer的buffer head。

为了增强系统性能,内核管理了一个小磁盘caches的bh_lrus数组,每个成员对应与一个CPU,被称为Least Recently Used (LRU) block cache。每个磁盘cache包含了8个指向buffer heads的指针,这些buffer heads是被对应CPU近期访问过的。

15.2.6.1. The __find_get_block( ) function

__find_get_block()函数接收参数:block_device描述符bdev,块号block,块大小size。返回buffer head地址,或NULL。函数执行了下列动作:

1. 检测当前CPU的LRU block cache,是否有需要的buffer head。
2. 如果buffer head在LRU block cache中,内核刷新数组,使刚发现的buffer head放到数组的index 0处,增加buffer head的b_count,跳转到步骤8。
3. 这一步,buffer head不在LRU block cache:内核从块号和块大小计算出页面index:

```
index = block >> (PAGE_SHIFT - bdev->bd_inode->i_blkbits);
```

4. 调用find_get_page()来定位包含了所需的block buffer的buffer page描述符。
5. 函数得到了buffer page描述符地址:它扫描链接在buffer page上的buffer heads链表,寻找所需的块。
6. 递减page->count。
7. 把LRU block cache中的所有元素向后移动,把刚访问的buffer head放在第一个位置。如果一个buffer head被移出了LRU block cache,它的b_count被减少。
8. 如果有必要,调用mark_page_accessed()来移动buffer page到合适的LRU链表。
9. 返回buffer head指针。

15.2.6.2. The __getblk( ) function

__getblk()函数绝不会失败:即使block根本不存在,函数分配一个bloxk device buffer page并且返回描述了该block的buffer head的指针。注意到函数返回的block buffer不一定包含了合法的数据。

__geetblk()函数执行了下列步骤:

1. 调用__find_get_block()来检查block是否在page cache中。如果block被找到了,函数返回它的buffer head地址。
2. 否则,调用grow_buffers()来为请求的block分配一个新的buffer page。
3. 如果grow_buffers()分配page失败, __geetblk()函数调用free_more_memory()来尝试回收一些内存。
4. 跳转到步骤1。

15.2.6.3. The __bread( ) function

__bread()函数与__getblk()相反,如果有必要,它在返回buffer head指针前,从磁盘中读取block。__bread()函数执行下列步骤:

1. 调用__getblk()得到buffer head的指针。
2. 如果block在page cache中并且buffer包含了合法的数据(BH_Uptodate设置了),返回buffer head地址。
3. 否则增加buffer head使用计数。
4. 设置b_end_io为end_buffer_read_sync()。
5. 调用submit_bh()传输buffer head到generic block layer。
6. 调用wait_on_buffer()把当前进程放入等待队列,直到读I/O操作完成。

7. 返回buffer head的地址。

15.2.7. Submitting Buffer Heads to the Generic Block Layer

submit_bh()和ll_rw_block()函数允许内核开始一个I/O数据传输,作用在用buffer heads描述的一个或多个buffers。

15.2.7.1. The submit_bh( ) function

内核利用submit_bh()函数传输一个buffer head给generic block layer,来请求一个单一数据块的传输。它的参数是:传输方向和指向buffer head的指针bh。

submit_bh()函数假设buffer head已被完全初始化;其中,b_bdev,b_blocknr,和b_size必须恰当的设置,用来指明磁盘中包含请求数据的块。如果block buffer属于一个block device buffer page,buffer head的初始化是通过__find_get_block()完成。

submit_bh()函数从buffer head的内容中创建了一个bio请求并且调用generic_make_request()。submit_bh()函数主要的执行步骤是:

1. 设置buffer head的BH_Req flag来表明block已经被至少提交一次;而且,如果数据传输方向是WRITE,清除BH_Write_EIO flag。
2. 调用bio_alloc()来分配一个新的bio描述符。
3. 根据buffer head的内容来初始化bio描述符:

a. 设置bi_sector为(bh->b_blocknr * bh->b_size / 512);
b. 设置bi_bdev为bh->b_bdev。
c. 设置bi_size为bh->b_size。
d. 初始化bi_io_vec数组的第一个元素。
e. 设置bi_vcnt为1(只有一个segment),bi_idx为0(当前要被传输的segment)。
f. 设置bi_end_io为end_bio_bh_io_sync(),设置bi_private为buffer head地址;当数据传输结束后, bi_end_io函数将被调用。

4. 增加bio的引用计数。
5. 调用submit_bio(),设置bi_rw为数据传输的方向,更新page_states这个per-CPU变量来跟踪读和写的sectors号。接着调用generic_make_request()函数。
6. 递减bio的使用计数;bio描述符不是空闲的,因为它现在已插入了I/O scheduler的队列。
7. 返回0(代表成功)。

当I/O数据传输结束时,内核执行bio的bi_end_io方法,在本节中,它实际调用的是end_bio_bh_sync()函数。submit_bh()函数本质上从bio中得到了buffer head地址,接着调用buffer head的b_end_io方法,最后调用bio_put()来销毁bio结构。

15.2.7.2. The ll_rw_block( ) function

有时内核必须一次触发多个数据块的传输,它们并不要求物理上邻近。ll_rw_block()函数接收参数:数据传输方向,要传输的块数量,一个buffer heads的指针数组。函数迭代的访问所有的buffer heads;对每个buffer head,它执行下列动作:

1. 测试和设置buffer head的BH_Lock flag;如果buffer已经被locked,说明数据传输已经被另一个内核控制流激活了,因此只需略过buffer并跳转到步骤9。
2. 增加buffer head的使用计数。

3. 如果数据传输方向是WRITE，设置buffer head的b_end_io方法为end_buffer_write_sync()函数的地址；否则，设置b_end_io方法为end_buffer_read_sync()函数的地址。
4. 如果数据传输方向为WRITE，将测试并清除buffer head的BH_Dirty flag。如果flag没有设置，那么不需要把block写入磁盘，因此跳转到步骤7。
5. 如果数据传输方向是READ或READA(read-ahead)，它检测buffer head的BH_Uptodate flag是否被设置；如果设置了，那么不需要从磁盘读取block，因此跳转到步骤7。
6. 这里block必须读或写：它调用submit_bh()函数把buffer head传给generic block layer，接着跳转到步骤9。
7. 清除BH_Lock来解锁buffer head，并且唤醒等待进程。
8. 递减buffer head的b_count。
9. 接着迭代处理下一个buffer head。

注意到如果ll_rw_block()函数传递一个buffer head给generic block layer，它将锁住buffer并递减计数器，因此buffer不能被访问也不能被释放，直到数据传输完成。当数据传输完成时，内核执行buffer head的b_end_io方法。假设没有发生I/O错误，b_end_io方法简单的设置buffer head的BH_Uptodate，解锁buffer，并递减它的使用计数。

15.3. Writing Dirty Pages to Disk

在下面几种条件下，dirty页面被写入磁盘：
1. page cache变得太满并且需要更多页面，或dirty页面数量变得太多。
2. 一个页面变成dirty状态时间太长。
3. 进程通过sync()，fsync()或fdatasync()系统调用来主动刷新。

当buffer page中只要有一个buffer head的BH_dirty置位，buffer page 的PG_dirty将置位。当内核选择一个dirty buffer page刷新，它将扫描与之关联的buffer heads并且只把dirty块写入磁盘。一旦内核刷新了buffer page中的所有dirty块，它将清除页面的PG_dirty。

15.3.1. The pdflush Kernel Threads

较早版本的Linux使用bdflush内核线程来扫描page cache寻找要刷新的dirty页面，利用第二个内核线程kupdate来确保没有页面在dirty状态停留太长时间。Linux2.6用一组名为pdflush的通用目的内核线程来代替这两个线程。

这些内核线程有一个灵活的结构。他们有两个参数：线程要执行的函数指针，函数所用的参数指针。系统中pdflush内核线程的数量是动态确定的：它们太少时会创建新线程，太多时会杀死已有线程。因为这些线程执行的函数可以阻塞，因此创建许多pdflush内核线程而不是一个单独的线程，将提高系统性能。

pdflush内核线程的创建与销毁遵循下列规则：
1. 最少两个pdflush内核线程，最多8个。
2. If there were no idle pdflush during the last second, a new pdflush should be created。

3. If more than one second elapsed since the last pdflush became idle, a pdflush should be removed。

每个pdflush内核线程有一个pdflush_work描述符。idle pdflush内核线程的描述符被收集到pdflush_list;pdflush_lock spin lock用来保护链表不被多处理器同时访问。nr_pdflush_threads变量保存了pdflush内核线程的总数。最后,last_empty_jifs变量保存了最后一次pdflush_list为空时的时间(in jiffies)。

每个pdflush内核线程执行__pdflush()函数,它本质上是一个无限循环直到内核线程死去。我们假设pdflush内核线程是idle;那么,进程进入睡眠,状态为TASK_INTERRUPTIBLE。一旦pdflush内核线程被唤醒,__pdflush()访问它的pdflush_work描述符并执行回调函数pdflush_work->fn,参数为pdflush_work->arg0。当回调函数结束时,__pdflush()检查last_empty_jifs变量:如果没有pdflush内核线程idle超过1秒并且pdflush内核线程数量小于8,__pdflush()开始另一个线程。否则m如果pdflush_list链表中最后一个线程idle超过1秒并且pdflush内核线程数量超过2,__pdflush()结束。否则,__pdflush()把它的pdflush_work描述符重新插入pdflush_list链表并且使内核线程进入睡眠。

pdflush_operation()函数用来激活一个idle pdflush内核线程。函数有2个参数:一个函数指针fn,函数指针的参数arg0;它执行下列步骤:

1. 从pdflush_list链表中提取指向一个idle pdflush内核线程的pdflush_work描述符的指针pdf。如果链表为空,它返回-1。如果链表只有1个元素,它设置last_empty_jifs变量为jiffies。
2. 把参数fn和arg0保存在pdf->fn,pdf->arg0。
3. 调用wake_up_process()唤醒idle pdflush内核线程,即: pdf->who。

pdflush通常执行下列回调函数之一来刷新dirty数据:

- background_writeout():系统的遍历page cache来查找需要被刷新的dirty页面。
- wb_kupdate():确保page cache中没有页面在dirty状态停留太长时间。

15.3.2. Looking for Dirty Pages To Be Flushed

每棵radix tree包含了要刷新的dirty页面。找到所有的dirty页面需要彻底的搜索所有的inodes的address_space对象。因为page cache可能包含了大量页面,在单一的一个控制流中搜索整个cache可能使CPU和磁盘忙碌较长时间。因此,Linux把page cache扫描分割成多个执行流。

wakeup_bdflush()函数接收参数:page cache中需要被刷新的dirty页面数量;如果参数为0,意味着所有dirty页面应该被写回磁盘。函数调用pdflush_operation()唤醒一个pdflush内核线程来执行background_writeout()回调函数。回调函数有效的把指定数量的页面写回磁盘。

wakeup_bdflush()函数当内存不足时或用户显式请求一个flush操作时将被调用。特别的,当下列情况时,这个函数被调用:

1. 用户进程发出sync()系统调用。
2. grow_buffers()函数分配一个新buffer page失败。
3. page frame回收算法调用free_more_memory()或try_to_free_pages()。

4. mempool_alloc()函数分配一个新的内存池中的元素失败。

此外,当一个进程修改了page cache中页面内容并且使dirty页面比例超过dirty background threshold,它会唤醒一个执行background_writeout()回调函数的pdflush内核线程。background threshold典型的被设置为系统所有页面的10%,它的值能通过写文件/proc/sys/vm/dirty_background_ratio来调整。

background_writeout()函数依赖于一个writeback_control结构,这个结构担当了一个双向通信设备:一方面,它告诉辅助函数writeback_inodes()做什么;另一方面,它保存了关于被写回磁盘的页面数量的统计值。这个结构最重要的成员有:

1. sunc_mode:指定同步模式:WB_SYNC_ALL意味着如果碰见一个被锁住的inode,它必须等待,不能略过;WB_SYNC_HOLD意味着锁住的inodes被放在一个链表中,以后来考虑;WB_SYNC_NONE意味着锁住的inodes被简单的略过。
2. bdi:如果它不是NULL,它指向一个backing_dev_info结构;在这种情况下,只有属于指定块设备的dirty页面会被刷新。
3. older_than_this: If not null, it means that inodes younger than the specified value should be skipped。
4. nr_to_write:在这个执行流中将要被写入磁盘的dirty页面数目。
5. nonblocking:如果该位被设置,进程不能被阻塞。

background_writeout()函数只有一个参数:_min_pages,应该刷新的页面的最小数量。它本质上执行下列步骤:

1. 从page_state这个per-CPU变量中读取当前保存在page cache中的页面数量和dirty页面数量。如果dirty页面的比例低于给定阈值并且至少_min_pages被刷新进了磁盘,函数结束。阈值典型的被设置为系统中页面总数的40%;可以通过写文件/proc/sys/vm/dirty_ratio,来调整阈值。
2. 调用writeback_inodes()来试图写1024个dirty页面。
3. 检测被有效写入的页面数量并且递减将要被写入的页面数量。
4. 如果少于1024个页面被写入或者如果页面被略过了,可能块设备的请求队列发生了拥塞:函数把当前进程放入一个特殊等待队列睡眠100毫秒或者直到队列变成非拥塞。
5. 跳转到步骤1。

writeback_inodes()函数只有一个参数:wbc指针指向一个writeback_control描述符。writeback_control->nr_to_write包含了将要被刷新进磁盘的页面数目。当函数返回时,这个成员包含了还未刷新的页面数量;如果一切顺利完成了,这个成员被设置为0。

我们假设writeback_inodes()被调用时,wbc->bdi和wbc->older_than_this指针被设为NULL,WB_SYNC_NONE同步模式,wbc->nonblocking设置。函数扫描了以super_blocks变量为起始的超级块链表。当整个链表被遍历完或目标数量的页面被刷新时,扫描结束。对每个超级块sb,函数执行下列步骤:

1. 检查sb->s_dirty或sb->s_io链表是否为空:第一个链表收集了超级块的dirty inodes,第二个链表收集了正在等待被传输到磁盘的inodes。如果2个链表都为空,文件系统的inodes没有dirty页面,函数将考虑链表中的下一个超级块。
2. 这里,超级块有dirty inodes,于是调用sync_sb_inodes()。函数执行下列动作:

a. 把sb->s_dirty链表里的inodes放入sb->s_io链表并且清空dirty inodes链表。
b. 从sb->s_io链表中得到下一个inode指针。如果链表为空,则返回。
c. 如果inodes是在sync_sb_inodes()开始后才变成dirty,它略过这些inodes的dirty页面并且返回。注意到,一些dirty inodes可能还留在sb->s_io链表中。
d. 如果当前进程是一个pdflush内核线程,它检查是否有另一个在另一个CPU上运行pdflush内核线程,已经在尝试刷新属于该块设备的dirty页面。这个检测能通过一个原子测试并设置操作inode->backing_dev_indo->BDI_pdflush flag。本质上,有超过1个pdflush内核线程作用在同样的请求队列中是无意义的。
e. 增加inode的使用计数。
f. 调用__writeback_single_inode()把与选定的inode相关联的dirty buffer写回。
    - 如果inode被锁定了,它把inode移动到dirty inodes链表(inode->i_sb->s_dirty)并且返回0。(因为我们之前假设wbc->sync_mode不是WB_SYNC_ALL,函数不会阻塞等待inode解锁)。
    - 使用inode的address_space的writepages方法,或者如果该方法不存在,则使用mpage_writepages()。这个函数使用find_get_pages_tag()函数来得到inode的address_space中的所有dirty页面。
    - 如果inode是dirty,它使用超级块的write_inode方法把inode写入磁盘。函数通常是依赖submit_bh()来实现这个方法,它传输一个单一的数据块。
    - 检测inode的状态;如果inode的某些页面仍然是dirty,就把inode移回到sb->s_dirty链表中,或者如果inode的引用计数是0,则移动到inode_unused链表中,否则移动到inode_in_use链表中。
    - 返回writepages中调用的函数返回的错误码。
g. 回到sync_sb_inodes()函数。如果当前进程是pdflush内核线程,它清除BDI_pdflush flag。
h. 如果在刚处理的inode中有一些页面被忽略了,那么inode包含了locked bffers:把所有sb->s_io中的inodes移动到sb->s_dirty链表:它们将在以后再被考虑。
i. 把inode使用计数减一。
j. 如果wbc->nr_to_write大于0,跳回至步骤2b来寻找同一个超级块的其它dirty inodes。否则,sync_sb_inodes()函数结束。

回到writeback_inodes()函数。如果wbc->nr_to_write大于0,跳转到步骤1并且接着处理全局链表中下一个超级块。否则,函数返回。

15.3.3. Retrieving Old Dirty Pages

如果一个页面在dirty状态超过一个预先定义的时间,内核将开始一个I/O数据传输,把页面内容写入磁盘。

获取old dirty pages的工作是由一个周期性被唤醒的pdflush内核线程完成。在内核初始化期间,page_writeback_init()函数设置了wb_timer动态定时器,定时器在dirty_writeback_centisecs个(通常是500)百分之一秒后超时。定时器超时后会调用wb_timer_fn(),它会唤醒pdflush_operation()函数,并把wb_kupdate()函数地址传给pdflush_operation()做参数。

wb_kupdate()函数扫描page cache来寻找"old" dirty inodes;它执行下列步骤:

a. 调用sync_supers()函数把dirty超级块写入磁盘。虽然这一步并不和刷新page cache中页面严格相关,但是这个函数调用确保了没有超级块会在dirty状态停留超过5秒。
b. 在writeback_control->older_than_this中保存指向一个变量的指针,这个变量保存了jiffies,等价于当前时间减去30秒。30秒是一个页面维持在dirty状态的最长时间。
c. 从per-CPU变量page_state中确定当前在page cache中的dirty页面数量。
d. 反复调用writeback_inodes()直到被写入磁盘的页面数目达到步骤3确定的值,或者所有dirty状态超过30秒的页面被写入磁盘。再循环中,如果请求队列变成拥塞,函数会进入睡眠。
e. 使用mod_timer()重启wb_timer动态定时器:它会在wb_kupdate()函数调用后,dirty_writeback_centisecs个百分之一秒后再次超时(或者如果这次执行太长了,那么将在1秒后超时)。

Chapter 16. Accessing Files

有多种方法来访问一个文件。本章中,我们将考虑下列情况:
1. Canonical mode:文件被打开时,O_SYNC和O_DIRECT flags清除,并且文件内容使用过read()和write()系统调用访问。在这种情况下,read()系统调用阻塞了调用进程,直到数据被拷贝到用户地址空间。write()系统调用则不同,因为它当把数据拷贝到page cache(defferd write)后就结束了。
2. Synchronous mode:文件被打开时,O_SYNC flag置位或者该flag被fcontl()在以后置位。这个flag仅仅影响write操作(读操作总是阻塞的),它阻塞了调用进程,直到数据被有效的写入磁盘。
3. Memory mapping mode:在打开文件后,应用程序发出mmap()系统调用来把文件映射到内存。结果是,文件在RAM中以一个字节数组形式出现,应用程序可已直接访问数组元素。
4. Direct I/O mode:文件被打开时,O_DERECT flag置位。任何读或写操作把数据直接从用户态地址空间传输到磁盘,反之亦然。它绕过了page cache。
5. Asynchronous mode:数据传输请求不会阻塞调用进程; they are carried on "in the background" while the application continues its normal execution。

16.1. Reading and Writing a File

读一个文件通常是page-based:内核总是一次性传输整个页面。如果一个进程发出一个read()系统调用来获取一些字节,但这些数据不在RAM中,内核分配一个新的page frame,用文件里的内容填充它,把页面添加到page cache,最后把请求的字节拷贝到进程地址空间。对绝大多数文件系统来说,从一个文件中读一个页面数据仅仅是在磁盘上寻找一个包含请求数据的磁盘块号。一旦这个动作完成,内核将提交合适的I/O操作给generic block layer来填充页面。实际上,所有的基于磁盘的文件系统的read方法是使用通用函数generic_file_read()实现的。

基于磁盘文件的写操作的处理稍微复杂些,因为文件大小可以增加,因此内核可以在磁盘上分配物理块。许多基于磁盘的文件系统是通过一个通用函数generic_file_write()来

实现它们的write方法。例如Ext2,System V/Coherent/Xenix,和MINIX。其它的一些文件系统则是使用专用的函数来实现write方法。

16.1.1. Reading from a File

绝大多数文件系统用generic_file_read()函数来实现read方法。函数有下列参数:
filp:file对象的地址。
buf:用户内存区域的线性地址,用来保存从文件中读取的字节。
count:要读取的字节数量。
ppos: Pointer to a variable that stores the offset from which reading must start。

第一步,函数初始化2个描述符。第一个描述符保存在局部变量local_iov中,数据类型是iovec;它包含了用户空间用来接收数据的buffer的地址(buf)和长度(count)。第二个描述符保存在局部变量kiocb中,数据类型是kiocb;它用来追踪同步或异步I/O操作的完成。

generic_file_read()函数通过执行init_sync_kiocb宏来初始化kiocb描述符,这个宏设置kiocb对象为同步操作。具体的,这个宏设置:kiocb->ki_key为KIOCB_SYNC_KEY, kiocb->ki_filp为filp, kiocb->ki_obj为current。

generic_file_read()函数调用__generic_file_aio_read(),并把iovec和kiocb作为参数传给它。__generic_file_aio_read()返回从文件中有效读取的字节数; generic_file_read()返回该值并结束。

__generic_file_aio_read()函数是一个通用目的的程序,被所有文件系统用来实现同步和异步读操作。这个函数接收4个参数:kiocb描述符地址iocb,iovec描述符数组地址iov,这个数组长度,文件当前指针ppos。当这个函数是被generic_file_read()调用时,iovec描述符数组只由一个元素组成,它描述了用户态用来接收数据的buffer。

现在我们解释__generic_file_aio_read()函数的动作;我们考虑最普遍的情况:一个由read()系统调用发出的作用在page-cached 文件上的同步操作。

这里是函数执行的步骤:
1. 调用access_ok()检查iovec描述符描述的用户态buffer是否有效。如果参数是无效的,返回-EFAULT错误码。
2. 建立一个读操作描述符,数据结构类型为read_descriptor_t,它保存了正在进行的读操作的当前状态。
3. 调用do_generic_file_read()。
4. 返回拷贝到用户态buffer的字节数;这个值可以在read_descriptor_t->written中找到。

do_generic_file_read()函数从磁盘中读取请求的页面并且把它们拷贝进用户态buffer。函数执行下列动作:
1. 读取要被读取的文件的address_space对象;它的地址在filp->f_mapping中。
2. 得到address_space对象的owner,也就是inode对象;它的地址保存在address_space->host中。如果被读取的文件是一个块设备文件,owner是bdev特殊文件系统中的inode,而不是由filp->f_dentry->d_inode指向的inode。
3. 计算出请求的数据的第一个字节所在页面的index和offset。

4. 循环读取包含了请求数据的所有页面；要被读取的字节数保存在read_descriptor_t->count中。每次迭代，函数传输一个页面的数据，它执行下列子步骤：

a. 如果index*4096+offet超过了文件大小，从循环退出并跳转到步骤5.

b. 调用cond_resched()来检查当前进程的TIF_NEED_RESCHED flag，如果置位了，调用schedule()函数。

c. 如果额外的页面必须提前读取，调用page_cache_readahead()。

d. 调用find_get_page()；函数查询page cache来寻找包含了请求数据的页面描述符。

e. 如果find_get_page()返回NULL，那么请求的页面不再page cache中。这种情况下，函数执行下列动作：

1.调用handle_ra_miss()来调整使用在read-ahead中的参数。

2.分配一个新页面。

3.调用add_to_page_cache()把新页面的描述符插入page cache。这个函数把新页面的PG_locked置位。

4.调用lru_cache_add()把新页面描述符插入到LRU链表。

5.跳转到4j开始读取文件数据。

f. 如果函数执行到这里，说明页面在page cache中。检查PG_uptodate flag；如果设置了，保存在页面上的数据是最新的，那么不需要从磁盘中读取数据：跳转到4m。

g. 页面中的数据不合法，因此它必须从磁盘中读取。函数调用了lock_page()来独占的访问页面。如果PG_locked已经设置， lock_page()会把当前进程挂起，直到这个bit被清0。

h. 现在页面被当前进程锁住了。然而，另一个进程可能已经把页面从page cache中移走了；那么，它检查页面描述符的mapping是否为NULL；如果是，它调用unlock_page()解锁页面，递减它的使用计数器，跳转回步骤4a。

i. 如果函数到了这里，页面被锁定了并且仍然在page cache中。再次检测PG_uptodate flag，因为另一个内核控制流可能在4f和4g之间就完成了必要的读操作。如果flag被设置，它调用unlock_page()并且跳转到4m跳过读操作。

j. 现在真正的I/O操作能开始了。调用address_space->readpage方法。这个方法对应的函数激活了从磁盘到页面的I/O数据传输。

k. 如果PG_uptodate flag仍然是清0的，它调用lock_page()等待页面被有效的读入。在步骤4中被加锁页面，一旦读操作完成就会被解锁。所以，当前进程会睡眠直到I/O数据传输结束。

l. 如果index超过文件大小，它递减页面的使用计数，退出循环跳转到步骤5。这种情况发生在当文件正在被读取时，同时被另一个进程缩小了。

m. 保存需要拷贝到用户态buffer中的页面中的字节数。这个值等于页面大小，除非offset不等于0。这发生在页面是第一个或最后一个。

n. 调用mark_page_accessed()来设置PG_referebced或PG_active flag，它们表示页面正在被使用并且不能被换出。

o. 现在是时候拷贝页面中的数据到用户态buffer了。do_generic_file_read()调用file_read_actor()函数，这个函数的地址是作为参数传进来的。file_read_actor()执行了下列步骤：

1.如果页面是在高内存中，则调用kmap()建立一个permanent kernel映射。

2.调用__copy_to_user()。注意到这个操作可以阻塞进程，因为访问用户地址空间可能会发生页面错误。

3.调用kunmap()释放permanent kernel映射。

4.更新read_descriptor_t描述符的count,written和buf成员。
p. 根据实际传输到用户态buffer中的字节数来更新index和offset局部变量。
q. 递减页面描述符的使用计数。
r. 如果read_descriptor_t->count不等于0,说明还有数据要从文件中读取:跳转到步骤4a。
5. 所有请求的数据被读取了。函数更新filp->d_ra read-ahead数据结构。
6. 更新*ppos,使它文件中的下一个位置。
7. 调用update_atime()保存当前时间在文件的inode->i_atime中并且标记inode为dirty,返回。

16.1.1.1. The readpage method for regular files

address_space->readpage方法保存了函数地址,这个函数有效的激活了从物理磁盘到page cache的I/O数据传输。对常规文件, this field typically points to a wrapper that invokes the mpage_readpage( ) function.例如,Ext3文件系统的readpage方法是如下实现的:

```
int ext3_readpage(struct file *file, struct page *page)
{
    return mpage_readpage(page, ext3_get_block);
}
```

wrapper是必须的,因为mpage_readpage()函数接收参数: the descriptor page of the page to be filled and the address get_block of a function that helps mpage_readpage() find the right block. wrapper是文件系统相关的,并且提供了合适的函数来取得一个block。这个函数把相对于文件开始的block号转换为相对于磁盘的逻辑块号。当然,函数的第二个参数依赖与常规文件所属的文件系统;在前面的例子中,参数是ext3_get_block()函数。ext3_get_block()函数使用了一个buffer head保存精准信息:块设备(b_dev),请求的数据在设备中的位置(b_blocknr),块状态(b_state)。

mpage_readpage()函数在从磁盘中读取页面时,在两种不同策略中选择。如果包含了请求数据的block在磁盘上连续,那么函数使用一个单一的bio描述符,把读I/O操作提交给generic block layer。否则,每个块使用一个不同的bio描述符来读取。The filesystem-dependent get_block function plays the crucial role of determining whether the next block in the file is also the next block on the disk.

mpage_readpage()函数执行下列步骤:
1. 检测页面描述符的PG_private flag:如果设置了,页面是一个buffer page.这意味着页面已经从磁盘中读取,并且页面中的blocks在磁盘上不邻近:跳转到步骤11来一次一个块的从磁盘读到页面中。
2. 从page->mapping->host->i_blkbits得到块大小,计算2个变量:页面中保存的块的数量和页面中第一个块的文件块号(页面中第一个块相对于文件开始的index)。
3. 对页面中每一个块,调用作为参数传进来的get_block函数来或取逻辑块号,即,相对于磁盘开始的块索引。页面中的所有块的逻辑块号保存在一个局部数组中。
4. 检测异常条件。如果块在磁盘上不邻近,或一些块落入一个"file hole",或者一个block buffer已经被get_block函数填满,那么跳转到步骤11来一次一个块的从磁盘读到页面。

5. 如果函数到了这一步,所有的页面中的块在磁盘上是邻近的。然而,页面可能是文件中最后一个,那么页面中某些块在磁盘中没有镜像。如果是这样,函数用0来填充页面中对应的block buffer;否则,它设置页面描述符的PG_mappedtodisk。
6. 调用bio_alloc()分配一个新的bio描述符,它包含了一个单一的segment。初始化bio的bi_bdev和bi_sector为块设备描述符和页面中第一个块的逻辑块号。这两个信息在步骤3中已经确定了。
7. 设置bio的segment的bio_vec描述符为:页面的地址,要读取的第一个字节的偏移,要读取的字节总数。
8. 保存mpage_end_io_read()函数地址到bio->bi_end_io。
9. 调用submit_bio(),用数据传输的方向来设置bi_rw,更新page_states这个per-CPU变量来追踪读取的sectors数量,调用generic_make_request()函数。
10. 返回0(成功)。
11. 如果函数跳转到了这里,页面包含了在磁盘上未邻近的块。如果页面是最新的(PG_uptodate flag设置),函数调用unlock_page()解锁页面;否则,调用block_read_full_page()从一次一个块的从磁盘读到页面。
12. 返回0(成功)。

mpage_end_io_read()函数是bio完成时调用的方法。假设没有I/O错误,函数本质上设置了页面描述符的PG_uptodate flag,调用unlock_page()解锁页面并且唤醒睡眠进程,调用bio_put()销毁bio描述符。

16.1.1.2. The readpage method for block device files

常规文件的adress_space->readpage方法依赖于文件系统的类型,但是块设备文件的这个方法都是一样的。它用blkdev_readpage()函数实现,实际上调用的是block_read_full_page():

```
int blkdev_readpage(struct file * file, struct * page page)
{
    return block_read_full_page(page, blkdev_get_block);
}
```

可见blkdev_readpage()函数是一个block_read_full_page()函数的wrapper。block_read_full_page()函数的第二个参数是blkdev_get_block()地址,它把相对于文件的块号转换成相对于块设备开始的逻辑块号。对块设备文件,这两个块号实际上是相同的;因此,blkdev_get_block()函数执行下列步骤:

1. 检查页面中的第一个块号是否超过块设备的最后一个块号。如果大于,返回-EIO给写操作或0给读操作。
2. 设置buffer head的b_dev为bdev。
3. 设置buffer head的b_blocknr为作为参数传进来的文件块号。
4. 设置buffer head的BH_Mapped flag,表示b_dev和b_blocknr有效。

block_read_full_page()函数一次一个块的从磁盘中读数据到页面。这个函数用在从块设备文件中读取数据和常规文件中读取在磁盘上不邻近的块。它执行下列步骤:

1. 检查页面描述符的PG_private flag;如果它设置了,说明页面与一个buffer heads相联系。否则,函数调用create_empty_buffers()分配buffer heads。page->private指向页面的第一个buffer head。
2. 由page->index计算出页面中第一个块的文件块号。
3. 对页面中每个buffer head,执行下列子过程:

a. 如果BH_Uptodate flag设置,跳过这个buffer接着处理页面下一个buffer。

b. 如果BH_Mapped flag没有设置并且块没有超过文件末尾，调用依赖于文件系统的get_block函数。对常规文件，函数查询在磁盘上的文件系统的数据结构，找到buffer相对于磁盘开始的的逻辑块号。相反的，对于块设备文件，函数把文件块号看成逻辑块号。在两种情况下，函数把逻辑块号保存在buffer head的b_blockne成员中并设置BH_Mapped flag。
c. 再次测试BH_Uptodate flag，因为依赖于文件系统的get_block函数可以触发一个block I/O操作更新buffer。如果BH_Uptodate设置了，它处理页面的下一个buffer。
d. 把buffer head地址保存在arr局部数组中，并继续处理页面的下一个buffer。
4. 如果之前没有遇到文件空洞，函数设置页面的PG_mappedtodisk flag。
5. 现在arr局部数组中储存了一些buffer head地址，这些buffer内容不是最新的。如果arr数组是空的，那么页面中所有buffers是合法的。函数设置页面的PG_uptodate flag，调用unlock_page()解锁页面并且返回。
6. arr局部变量非空。对数组中每个buffer，block_read_full_page()执行下列子步骤：
a. 设置BH_Lock flag。如果flag已经被设置，函数将等待到buffer释放。
b. 设置buffer head的b_end_io成员为end_buffer_async_read()函数地址，并且设置buffer head的BH_Async_Read flag。
7. 对arr局部数组中每个buffer head，调用submit_bh()函数。这个函数触发I/O数据传输。
8. 返回0。

16.1.2. Read-Ahead of Files

许多磁盘访问是顺序的。保存在磁盘上的常规文件有很多组邻近的sectors。当一个程序读或拷贝一个文件，它常顺序访问文件，从第一个字节到最后一个字节。

在文件真正读取前，read-ahead读取许多邻近的常规文件或块设备文件的数据页面。在大多数情况下，read-ahead显著的增强了磁盘性能。而且，它提高了系统相应速度。一个顺序访问文件的进程通常不同等待请求的数据，因为数据已经在RAM中了。

然而，read-ahead对那些随机访问文件的应用是无作用的；在这种情况下，它实际上是有坏处的，因为它会在page cache中浪费空间。所以，内核当确认最近的I/O访问不是顺序的，那么减少read-ahead。

Read-ahead是一种很好的算法，因为：

- 因为数据是一个页面一个页面读取的， read-ahead算法不用考虑页面中的偏移，只需考虑访问的页面在文件中的位置。
- 当进程持续的顺序访问文件时，read-ahead会逐渐增加。
- 当进程访问文件不是顺序时，read-ahead必须减少或甚至禁止。
- 当进程持续访问同一个页面，或当文件几乎所有页面都在page cache中， read-ahead必须停止。
- 低级I/O设备驱动在合适的时间应该被激活，因此在进程需要某些页面前，它们早被传输进RAM。

当文件访问的第一个页面紧跟在上次访问文件时的最后一个页面，内核把文件访问看成是顺序的。

当访问一个给定文件,read-ahead算法利用了2个页面集合,每个集合都对应于文件中一个连续的部分。这两个集合被称为current window和ahead window。

current window由一些页面组成,这些页面被进程请求或是被内核提前读取,他们都在page cache中(一个在current window中的页面并不一定是最新的,因为它的I/O数据传输可能还在进行中)。current window包含了被进程最后访问过的页面和被内核需要提前读取的页面。

ahead window由一些页面组成,这些页面紧跟在current window中的页面之后,将被内核提前读取。ahead window中的页面并没有被进程请求,但是内核假设不久后进程将请求它们。

当内核确定了是顺序访问和属于current window的初始页面,它检查ahead window是否已经被建立了。如果没有,内核创建一个新的ahead window并且触发读操作。在理想情况下,进程在ahead window中的页面还在传输时,仍然从current window中请求页面。当进程请求一个属于ahead window中的页面时,ahead window变成新的current window。

被read-ahead算法实用的主要数据结构是file_ra_state描述符。每个file对象都有一个这种描述符成员f_ra。
当文件被打开,它的file_ra_state描述符除了prev_page和ra_pages外,其它成员都设置为0。

prev_page成员保存了进程在前一个读操作时最后请求的页面的index;初始的,这个成员包含值-1。

pa_pages成员代表了current window的最大值,也是最大read-ahead值。缺省值存储在块设备的backing_dev_info描述符中。

file_ra_state->flag成员有两种flags:RA_FLAG_MISS和RA_FLAG_INCACHE。当一个被提前读取的页面没有在page cache中找到时(可能是因为它被内核回收了),第一个flag被设置,在这种情况下,将要创建的下一个ahead window的大小被稍微减小。当内核确认进程请求的最后256个页面已经在page cache中时,第二个flag被设置。在这种情况下,read-ahead被关闭,因为内核假设进程所需的所有页面已经在cache中了。

read-ahead算法何时被执行呢?在下列情况下发生:

- 当内核处理一个读取文件数据的用户态请求时,它触发page_cache_readahead()函数调用。
- 当内核为一个文件内存映射分配一个页面。
- 当一个用户模式应用执行readahead()系统调用,它显式的触发一些read-ahead活动。
- When a User Mode application executes the posix_fadvise( ) system call with the POSIX_FADV_NOREUSE or POSIX_FADV_WILLNEED commands, which inform the kernel that a given range of file pages will be accessed in the near future.
- When a User Mode application executes the madvise( ) system call with the MADV_WILLNEED command, which informs the kernel that a given range of pages in a file memory mapping region will be accessed in the near future.

16.1.2.1. The page_cache_readahead( ) function

page_cache_readahead()函数实现了所有read-ahead相关操作。它补充current和ahead windows,根据read-ahead hits数量来更新它们的大小。

这个函数当内核必须满足一个读请求时才执行,它有5个参数:
1. mapping: 指向address_space对象。
2. ra:指向file_ra_state描述符。
3. filp:file对象的地址。
4. offset:页面在文件中的偏移。
5. req_size:需要读取的页面数量。

当进程第一次访问文件并且请求的第一个页面在文件中的偏移是0,函数假设进程将执行顺序访问。那么,函数创建一个新的currnet window。初始的current window长度总是2的幂,这个长度与进程第一次读操作请求的页面数量有关:请求的页面越多,current window就越大,其最大值保存在ra->ra_pages。相反地,当进程第一次访问文件,但请求的第一个页面在文件中的偏移不是0,函数就假设进程不执行顺序访问。那么,函数临时的关闭read-ahead。而且,当函数认为它是顺序访问时,一个新的current window将被创建。

如果ahead window已经不存在了,一旦函数认为进程在current window中执行一个顺序访问时,ahead window将被创建。ahead window总是从current window中的最后一个页面开始。它的长度,与current window的长度相关:如果RA_FLAG_MISS flag被设置,ahead window的长度是current window的长度减2,或如果小于4,则等于4个页面;否则,ahead window的长度是current window的长度的4倍或2倍。如果进程继续顺序的访问文件,最后ahead window变成新的current window,并且一个新的ahead window被创建。那么,read-ahead是大大增强了进程顺序读文件的速度。

一旦函数认为一个文件访问不是顺序的, current window和ahead window被清空,并且read-ahead被临时禁止。Read-ahead会在进程执行顺序访问时再次重启。

每次page_cache_readahead()创建一个新window,它开始读操作。为了读取a chunk of pages,page_cache_readahead()调用blockable_page_cache_readahead()函数。为了减少内核开销,后一个函数选择下列方法:

- 如果服务块设备的请求队列是read-congested,那么读操作不会执行。
- 再次检测page cache中每个要读的页面;如果页面已经在page cache中,它只是简单的略过。
- 请求队列所需的所有page frames在执行从磁盘读操作前被分配。如果不是所有page frame都被得到,那么read-ahead操作只会在可用pages上执行。
- 只要可能,将通过multi-segment bio描述符来把读操作提交给generic block layer。如果有定义,这个动作由address_space->readpages方法完成;否则,将反复调用address_space->readpage方法。

16.1.3. Writing to a File

write()系统调用把数据从用户态地址空间移动到内核数据结构中,然后移入磁盘。file对象的write方法允许每个文件系统定义一个特殊的写操作。在Linux2.6,每种基于磁盘

的文件系统的write方法，识别在写操作中的磁盘块号，拷贝数据到page cache中的一些页面中，标记这些页面中的buffers为dirty。

许多文件系统实现file对象的write方法是通过generic_file_write()函数，它执行下列步骤：

1. 初始化一个iovec类型的局部变量，它包含用户态buffer的地址和长度。
2. 把被写入的文件的inode对象的地址存入inode变量中，并请求semaphore inode->i_sem。
3. 调用init_sync_kiocb宏来初始化kiocb类型的局部变量。
4. 调用__generic_file_aio_write_nolock()标记受影响的页面为dirty。
5. 释放inode->i_sem semaphore。
6. 检测文件的O_SYNC flag，inode的S_SYNC flag，和超级块的MS_SYNCHRONOUS flag；如果它们中至少有一个被设置，那么调用sync_page_range()函数强迫内核刷新page cache中的所有页面，阻塞当前进程直到I/O数据传输结束。接着，sync_page_range()执行address_space对象的writepages方法(如果有定义)，或mpage_writepages()函数来开始对dirty页面的I/O传输；接着，它调用generic_osync_inode()刷新inode和相关buffers到磁盘，最后调用wait_on_page_bit()挂起当前进程，直到所有被刷新的页面的PG_writeback被清0。
7. 返回被有效写入磁盘的字节数目。

__generic_file_aio_write_nolock()函数最常见的情况是被write()系统调用使用，在这种情况下，函数执行下列动作：

1. 调用access_ok()来检测用户态buffer是否有效。如果参数为非法，它返回-EFAULT错误码。
2. 把被写入的文件的inode对象的地址存入inode变量中。注意到，如果文件是个块设备，那么这个inode是在bdev特殊文件系统中。
3. 设置current->backing_dev_info为文件的backong_dev_info描述符的地址。本质上，这个设置允许当前进程把dirty页面写回，甚至对应的请求队列是拥塞的。
4. 如果file->flags的O_APPEND flag被设置并且文件是常规的（不是块设备文件），设置*ppos为文件末尾，使新数据被加在后面。
5. 对文件大小进行检测，它不能超过一些限制。
6. 如果设置了，则清0 suid flag；如果文件是可执行的，则清0 sgid flag。
7. 把当前时间保存在inode->mtime和inode->ctime中，并且标记inode对象为dirty。
8. 开始一个循环来更新在写操作中涉及的文件页面。每次迭代中，它执行下列步骤：

a. 调用find_lock_page()在page cache中搜索页面。如果找到了页面，它递增页面使用计数并设置PG_locked flag。
b. 如果页面不在page cache中，那么它分配一个新page frame，并调用add_to_page_cache()把页面插入page cache。新页面同时还被插入memory zone的不活跃链表。
c. 调用address_space->prepare_write方法。函数为页面分配并初始化buffer heads。
d. 如果buffer在高内存中，它建立了用户态buffer的一个内核映射。那么，调用__copy_from_user()把用户态buffer里的字节拷贝到页面，然后释放内核映射。

e. 调用address_space->commit_write方法。对应的函数标记buffer为dirty,它们将在之后被写入磁盘。
f. 调用unlock_page()清0 PG_locked flag并且唤醒正在等待该页面的进程。
g. 调用mark_page_accessed()更新页面的用于内存回收算法的状态。
h. 递减页面使用计数。
i. 检测page cache中dirty页面的比例是否超过一个固定值(通常是40%);如果是,调用writeback_inodes()把一些页面刷新进磁盘。
j. 调用cond_resched()检测当前进程的TIF_NEED_RESCHED flag,如果它被设置了,就调用schedule()函数。
9. 现在,所有写操作涉及的文件页面已经被处理了。更新*ppos。
10. 设置current->backing_dev_info为NULL。
11. 返回有效写入的字节数目,函数结束。

16.1.3.1. The prepare_write and commit_write methods for regular files

address_space对象的prepare_write和commit_write方法被generic_file_write()用来处理常规文件和块设备文件。这两个方法在处理每个页面时都会被调用一次。

每个基于磁盘的文件系统定义了它自己的prepare_write方法。像读操作那样,这个方法是一个通用函数的wrapper。例如,Ext2文件系统通常这样实现prepare_write方法:

```
int ext2_prepare_write(struct file *file, struct page *page,
                       unsigned from, unsigned to)
{
    return block_prepare_write(page, from, to, ext2_get_block);
}
```

ext2_get_block()函数把相对于文件的块号转换为逻辑块号。

block_prepare_write () 函数准备好了文件页面的buffers和buffer heads,它执行下列步骤:

1. 检测页面是否是buffer page(检测PG_private);如果flag被清0,调用creat_empty_buffers()为页面分配buffer heads。
2. 对每个buffer head,执行下列动作:
a. 复位BH_New flag。
b. 如果BH_Mapped flag没有被设置,函数执行下列步骤:
1.调用作为参数传进来的get_block函数。这个函数查看文件系统的磁盘上的数据结构,找到buffer的逻辑块号(相对于磁盘开始而不是相对于文件开始)。这个函数把块号存放在对应buffer head的b_blocknr成员中,并设置BH_Mapped flag。
2.Checks the value of the BH_New flag; if it is set, invokes unmap_underlying_metadata( ) to check whether some block device buffer page in the page cache includes a buffer referencing the same block on disk.如果找到了,函数清除BH_Dirty flag并等待作用在这个buffer上的I/O数据传输完成。如果写操作没有重写页面中的所有buffer,它会其它位写入的buffer置为0。
c. 如果写操作没有重写整个buffer,并且它的BH_Delay和BH_Uptodate flags没有设置(RAM中的buffer没有包含合法的磁盘镜像)。函数调用ll_rw_block()从磁盘中读取数据。
3. 阻塞当前进程,直到所有读操作完成。
4. 返回0。

一旦prepare_write方法返回后,generic_file_write()函数用在用户态的地址空间的数据来更新页面。接着,它调用commit_write方法。大多数基于磁盘的文件系统的这个方法是用generic_commit_write()来实现的。

generic_commit_write()函数执行了下列步骤:

1. 调用__block_commit_write()函数。这个函数执行下列:
   a. 调用写操作涉及的页面中的所有buffers;对每个buffer,设置BH_Uptodate和BH_dirty flags。
   b. 标记对应的inode为dirty。同时把,该inode加入超级块的dirty inodes链表。
   c. 如果页面中所有buffers是最新的,设置页面的PG_uptodate flag。
   d. 设置页面的PG_dirty flag,以及标记该页面在radix tree中的标签为dirty。
2. 检测写操作是否扩大了文件。更新文件的inode->i_size成员。
3. 返回0。

16.1.3.2. The prepare_write and commit_write methods for block device files

写块设备文件的操作与写常规文件的操作类似。块设备的address_space对象的prepare_write方法如下实现:

```
int blkdev_prepare_write(struct file *file, struct page *page,
                         unsigned from, unsigned to)
{
    return block_prepare_write(page, from, to, blkdev_get_block);
}
```

与之前作用在常规文件的prepare_write方法区别在于,文件块号和逻辑块号是相同的。

块设备的address_space对象的commit_write方法如下实现:

```
int blkdev_commit_write(struct file *file, struct page *page,
                        unsigned from, unsigned to)
{
    return block_commit_write(page, from, to);
}
```

与之前作用在常规文件的commit_write方法方法区别在于,不检测文件是否被扩大了,因为设备文件是不允许扩大的。

16.1.4. Writing Dirty Pages to Disk

当内核打算开始I/O数据传输,它调用文件的address_space对象的writepages方法,这个方法在radix-tree中搜索dirty pages并且刷新进磁盘。例如,Ext2文件系统如下实现了这个writepages方法:

```
int ext2_writepages(struct address_space *mapping,
                    struct writeback_control *wbc)
{
    return mpage_writepages(mapping, wbc, ext2_get_block);
}
```

mpage_writepages()函数本质上执行下列动作:

1. 如果请求队列是write-congested并且进程不想阻塞,函数不进行任何写入操作就直接返回。
2. 如果writeback_control描述符指定了文件中的初始位置,函数会把它转换为一个页面index。否则,如果writeback_control描述符指明进程不打算等待I/O数据传输完成,它把mapping->writeback_index的值作为初始页面index(也就是

说,扫描是从之前的写回操作的最后一个页面开始)。最后,如果进程必须等待I/O数据传输完成,扫描将从文件的第一个页面开始。

3. 调用finde_get_pages_tag()在page cache中的查找dirty页面描述符。
4. 对每个在步骤3中找到的页面描述符,函数执行下列步骤:

a. 调用lock_page()锁住页面。
b. 检测页面仍然是有效的并且还在page cache中(因为另一个内核控制流可能在步骤3和4a之间对页面进行了操作)。
c. 检测页面的PG_writeback flag。如果它被设置了,说明页面已经正在被刷新进磁盘中。如果进程必须等待I/O数据传输完成,它调用wait_on_page_bit()阻塞当前进程,直到PG_writeback flag被清0。否则,如果进程不想等待,它检测PG_dirty flag:如果它现在已清0,那么解锁页面并且跳回步骤4a继续处理下一个页面。
d. 如果get_block参数为NULL,它调用mapping->writepage方法把页面刷新进磁盘。否则,如果get_block参数非NULL,它调用mpage_writepage()函数。

5. 调用cond_resched()检测当前进程的TIF_NEED_RESCHED flag并且,如果flag被设置了,调用schedule()函数。
6. 如果函数没有扫描给定范围的所有页面,或者如果有效写入磁盘的页面数量小于writeback_control描述符指定的值,它跳转回步骤3。
7. 如果writeback_control描述符没有指定文件的初始位置,设置mapping->writeback_index为最后一次扫描的页面的index。
8. 如果mpage_writepage()函数已经在步骤4d中调用了,并且如果函数返回一个bio描述符地址,它调用mpage_bio_submit()。

一个典型的文件系统例如Ext2实现writepage方法,是一个a wrapper for the general-purpose block_write_full_page()函数。block_write_full_page()函数类似与block_read_full_page():它给页面分配buffer heads,调用submit_bh()。如果是对块设备文件的操作, writepage方法使用blkdev_writepage()实现的,这个函数wrapper for block_write_full_page().

许多non-journaling filesystems依赖与mpage_writepage()函数,而不是通用的writepage方法。这能够提高性能,因为mpage_writepage()函数在同一个bio描述符中收集尽可能多的页面;这种方法能使块设备驱动利用scatter-gather DMA。

16.2. Memory Mapping

一个memory region能与一个常规文件或一个块设备文件的一步份相联系。这一位置访问这个memory region将被内核转换为对文件的操作。这种技术被称为内存映射。

有2种类型的内存映射:
Shared:每个对映射了的内存的写操作将改变磁盘上的文件;而且,如果一个进程对共享内存映射了的页面的修改,将对其它映射着同一个文件的进程可见。
Private:被进程用来创建只读文件映射。对映射内存的写操作不会改变磁盘上的文件,也不会被其它映射了同一个文件的进程看见。However, pages of a private memory mapping that have not been modified by the process are affected by file updates performed by other processes.

一个进程能通过系统调用mmap()来创建一个内存映射。程序员必须指定MAP_SHARED flag或MAP_PRIVATE flag作为这个系统调用的参数。进程通过mumap()系统调用来撤销内存映射。

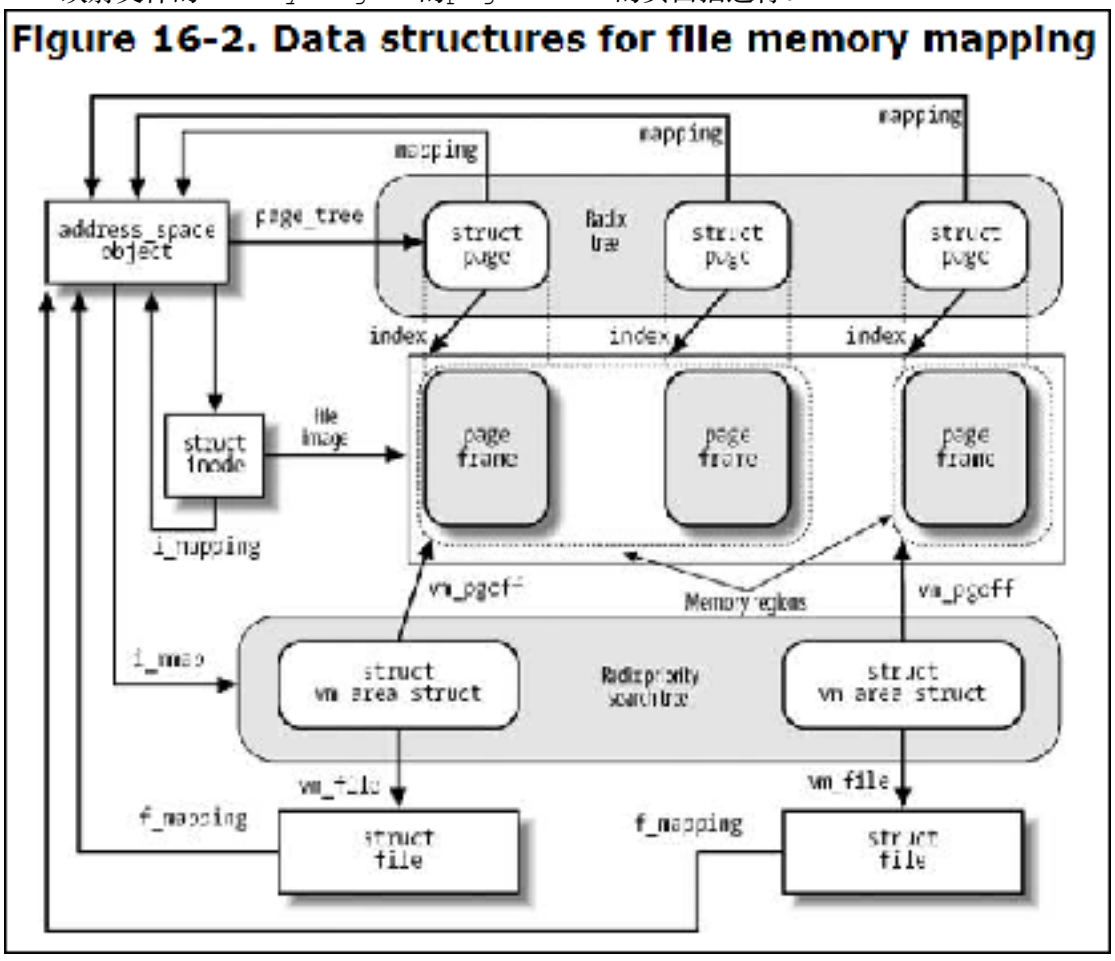
如果一个内存映射是共享的,对应的memory region的VN_SHARED flag 设置;如果它是私有的,VM_SHARED flag被清0。

16.2.1. Memory Mapping Data Structures

一个内存映射由下列数据结构的组合来代表:

- 与被映射了文件相联系的inode对象。
- 被映射的文件的address_space对象。
- 对同一个文件的不同映射,都各自有一个file对象。
- 每个文件映射都有一个vm_area_struct描述符。
- 映射文件的memory region的page frame的页面描述符。

Figure 16-2. Data structures for file memory mapping



上图说明了数据结构是如何链接在一起的。图左边是inode,它标识了文件。inode->i_mapping指向文件的address_space对象。address_space对象的page_tree成员指向属于该address space的页面radix tree,i_mmap成员指向属于该address space的memory region的radix priority search tree(PST)。PST的主要作用是为了执行"反向映射"。于同一个文件关联的多个file对象和inode之间的链接是通过f_mapping成员链在一起。

每个memory region描述符都有一个vm_file成员,使之链接到被映射文件的file对象(如果这个成员是NULL,说明memory region没用使用在一个内存映射中)。The

position of the first mapped location is stored into the vm_pgoff field of the memory region descriptor; it represents the file offset as a number of page-size units.映射的文件长度就是memory region长度,可以通过vm_srart与vm_end计算出来。

共享内存映射的页面总是在page cache中;私有内存映射的页面,当它们还未被修改前会一直在page cache中。当一个进程试图修改一个私有内存映射时,内核复制page frame并且在进程的Page Table中用赋值的替换原有的page frame。原始的page frame仍然留在page cache中,但是它不再属于内存映射,因为它已经被复制版本替换了。复值版本不会插入page cache,因为它包含的数据对于磁盘上的文件是无效的。

内存映射的核心部分是file对象的mmap方法。对大多数基于磁盘的文件系统和块设备文件,这个方法是通过通用函数generic_file_mmap()来实现的。

实际上,一个新创建的memory region没有包含任何页面;当进程引用region中的一个地址时,一个Page Fault发生, Page Fault处理函数检查memory region的nopage方法是否定义。如果nopage没有定义,说明memory region没有映射到磁盘上一个文件;否则,它有定义,这个方法实现了访问块设备读取页面。大多数基于磁盘的文件系统和块设备文件用filemap_nopage()来实现nopage方法。

16.2.2. Creating a Memory Mapping

为了创建一个内存映射,一个进程发出mmap()系统调用,它的参数是:

- 一个要被映射的文件的file描述符。
- 文件偏移量,指明了要被映射文件的初始位置。
- 要被映射的文件长度。
- 一个flags集合。进程必须显式设置MAP_SHARED flag或MAP_PRIVATE flag。
- 一个访问memory region的访问权限集合:读访问(PROT_READ),写访问(PROT_WRITE),或执行访问(PROT_EXEC)。
- 一个可选的线性地址,它被内核用来考虑作为新memory region开始地址。如果MAP_FIXED flag被设置并且内核不能从指定的线性地址开始分配一个新memory region,系统调用失败。

mmap()系统调用返回新memory region的开始位置的线性地址。

16.2.3. Destroying a Memory Mapping

当进程准备销毁一个内存映射,它调用munmap();这个系统调用也用来减少memory region的大小。

16.2.4. Demand Paging for Memory Mapping

因为效率原因,在内存映射被创建时,page frames并没有马上被分配,当最后时刻进程尝试访问内存映射的页面时,会发生一个Page Fault异常。

do_no_page()函数执行了各种类型的请求页面操作,例如分配一个page frame和更新Page Tables。它也检测memory region的nopage方法是否定义了。第9章的"Demand Paging"小节,我们描述的是nopage方法未定义的情况(anonymous memory region),现在我们考虑nopage方法定义了的情况:

1. 调用nopage方法,它返回包含了请求页面的page frame的地址。
2. If the process is trying to write into the page and the memory mapping is private, it avoids a future Copy On Write fault by

making a copy of the page just read and inserting it into the inactive list of pages (see Chapter 17). If the private memory mapping region does not already have a slave anonymous memory region that includes the new page, it either adds a new slave anonymous memory region or extends an existing one (see the section "Memory Regions" in Chapter 9). In the following steps, the function uses the new page instead of the page returned by the nopage method, so that the latter is not modified by the User Mode process.

3. 如果其它进程has truncated or invalidated the page(address_space的truncate_count成员被用在这种检测上),函数跳转到步骤1来重新取得页面。
4. 增加进程内存描述符的rss成员,表明一个新的page frame被分配给进程。
5. Sets up the Page Table entry corresponding to the faulty address with the address of the page frame and the page access rights included in the memory region vm_page_prot field.
6. 如果进程正在尝试写页面,它使对应Page Table entry的Read/Write和Dirty位置1.在这种情况下,page frame要么独占的分配给进程,或页面是共享的;在两种情况下,写页面都是允许的。

当处理的memory region映射了磁盘上的文件,nopage方法必须在page cache中搜索请求的页面。如果页面没找到,nopage方法必须从磁盘中读取它。大多数文件系统使用函数filemap_nopage()实现了该方法,他接收3个参数:

1. area:包含了请求页面的memory region地址。
2. address:请求页面的线性地址。
3. Pointer to a variable in which the function writes the type of page fault detected by the function (VM_FAULT_MAJOR or VM_FAULT_MINOR).

filemap_nopage()函数执行下列步骤:

1. 从area->vm_file得到file对象地址,保存在file中。从file->f_mapping中得到address_space对象地址。从address_space->host得到inode对象地址。
2. 使用area的vm_start和vm_pgoff成员确定address线性地址在文件中对应的数据块的偏移量。
3. 检测刚得到的偏移量是否超过了文件大小。如果超过了,返回NULL。
4. 如果memory region的VM_RAND_READ flag被设置,我将假设进程会随机的读取内存映射页面。在这种情况下,跳转到步骤10,忽略read-ahead。
5. 如果memory region的VM_SEQ_READ flag被设置,我们假设进程将顺序的读取内存映射页面。在这种情况下,函数调用page_cache_readahead()从faulty page处开始执行read-ahead。
6. 调用find_get_page()来查找page cache。如果页面被找到,跳转到步骤11。
7. 如果函数执行到这一步,页面没有在page cache中找到。检查memory region的VM_SEQ_READ flag:
   - 如果flag设置了,内核已经之前已提前读取了memory region的页面,然而并没有找到请求的页面,那么说明read-ahead算法失败了:它调用handle_ra_miss()调整read-ahead参数,接着跳转到步骤10。
   - 否则,如果flag被清0,它把文件的file_ra_state描述符的mmap_miss计数器加1。如果mmap_miss数量打与mmap_hit值,它将忽略read-ahead并跳转到步骤10。
8. 如果read-ahead没有被永久关闭(file_ra_state->ra_pages大于0),函数调用do_page_cache_readahead()读取请求页面附近的一个页面集合。

9. 调用find_get_page()检查请求的页面是否在page cache中;如果在,跳转到步骤11。
10. 调用page_cache_page()。这个函数检查请求页面是否在page cache中,如果不在则分配一个新的page frame,添加到page cache中,执行mapping->a_ops->readpage方法来从磁盘中读取页面内容。
11. Invokes the grab_swap_token( ) function to possibly assign the swap token to the current process.
12. 请求的页面现在在page cache中。把文件的file_ra_state描述符的mmap_hit计数器加一。
13. 如果页面不是最新的(PG_uptodate flag清0),它调用lock_page()锁住页面,执行mapping->a_ops->readpage方法触发I/O数据传输,调用wait_on_page_bit()进入睡眠直到数据传输结束。
14. 调用mark_page_accessed()标记请求的页面为被访问过。
15. 如果在page cache中找到的页面是最新的了,设置*type为VM_FAULT_MINOR;否则设置它为VM_FAULT_MAJOR。
16. 返回请求页面的地址。

16.2.5. Flushing Dirty Memory Mapping Pages to Disk

16.2.6. Non-Linear Memory Mappings

Linux2.6内核提供了另一种对常规文件的访问方法:非线性内存映射。一个对文件的非线性内存映射也是之前提到的文件映射的一种,但是内存页面并不是顺序的映射到文件页面;每个内存页面可以映射到文件中任意一个页面。

一个用户应用可以通过反复调用mmap()系统调用实现同样的效果。然而,这种方法效率比较低,因为每次映射都需要一个不同的memory region。

为了支持非线性内存映射,内核利用一些额外的数据结构。首先,memory region描述符的VM_NONLINEAR flag指明了memory region包含了一个非线性映射。所有的给定文件的非线性映射memory region描述符都被链入了address_space->i_mmap_nonlinear中。

为了创建一个非线性内存映射,用户态应用首先调用mmap()系统调用创建一个正常的内存映射。接着,应用程序调用remap_file_pages()系统调用重映射其中的某些页面。这个系统调用的服务程序sys_remap_file_pages()使用4个参数:

1. start: A linear address inside a shared file memory mapping region of the calling process.
2. size:文件的要重新映射的大小,以字节为单位。
3. prot:未使用(必须为0)。
4. pgoff:要被重新映射文件页面的初始index。
5. flags:控制非线性内存映射的flags。

服务程序从start线性地址开始重映射文件的某部分(由pgoff和size决定)。如果memory region不是共享的或它不够大来包含所有请求的页面,系统调用失败,一个错误码被返回。本质上,服务程序把memory reigon插入文件的i_mmap_nonlinear链表并且调用memory region的populate方法。

对所有常规文件,populate方法是通过filemap_populate()函数实现,它执行下列步骤:

1. 检查flags参数的MAP_NONBLOCK flag是否清0；如果，调用do_page_cache_readahead()来提前读取被重映射的文件页面。
2. 对每个被重映射的页面，执行下列步骤：

a. 检测页面描述符是否在page cache中；如果不再并且MAP_NONBLOCK flag被清0，它从磁盘中读取页面。
b. 如果页面描述符在page cache中，它更新Page Table entry，更新memory region描述符的页面计数器。
c. 否则，如果页面描述符没有在page cache中找到，它把文件页面偏移值保存在对应线性地址的Page Table entry的高32-bit；清0 Page Table entry的Present位并且设置Dirty位。

当处理一个缺页错误时，handle_pte_fault()函数检测Page Table entry的Present和Dirty位；如果它们对应了非线性内存映射，handle_pte_fault()调用do_file_page()函数， do_file_page()从Page Table entry的高32位提取文件偏移；接着do_file_page()调用memory region的populate方法从磁盘中读取页面并更新Page Table entry。

16.3. Direct I/O Transfers

在Linux2.6中，通过文件系统访问文件和通过块设备文件访问没有本质上的区别。一些复杂应用(self-caching applications)希望完全的控制整个I/O数据传输机制。例如，高性能数据库服务器：它们有自己的caching机制，充分利用了数据库查询的独特的性质。对这种程序，内核页面cache没有任何帮助；相反的，它对性能还有害，因为：

- 大量page frames被浪费在已经存在于RAM中的重复数据。
- read()和write()系统调用速度被多余的指令(处理page cache和read-ahead)降低了；文件内存映射相关的系统调用也有类似的问题。
- read()和write()系统调用有两次数据拷贝：磁盘到内核buffer，内核buffer到用户内存。

Linux提供了一种简单的方法来绕过page cache：直接I/O传输。In each I/O direct transfer, the kernel programs the disk controller to transfer the data directly from/to pages belonging to the User Mode address space of a self-caching application.

每次数据传输处理都是异步的。当处理在进行时，内核可能会切换当前进程，CPU返回用户态，产生数据传输的进程的页面可能会被swap out，等等。This works just fine for ordinary I/O data transfers because they involve pages of the disk caches . Disk caches are owned by the kernel, cannot be swapped out, and are visible to all processes in Kernel Mode.

另一方面，直接I/O传输会移动数据到用户空间的页面。内核必须确保这些页面能被内核进程访问，并且页面不能在数据传输还在进行时就被交换出去。让我们来看看这是怎么实现的。

当一个self-caching application打算直接访问一个文件，它指定了O_DIRECT flag并打开文件。内核对open()系统调用的服务程序，调用dentry_open()函数来检测

被打开文件的address_space对象的direct_IO方法是否定义了,如果没定义就返回一个错误码。

文件的read方法通常是用generic_file_read()函数实现的,它初始化了iovec和kiocb描述符,调用__generic_file_aio_read()。__generic_file_aio_read()函数验证用户态buffer是否合法,接着检测文件的O_DRECT flag是否被设置了。当我们执行read()系统调用,函数执行下列代码片断:

```
if (filp->f_flags & O_DIRECT) {
    if (count == 0 || *ppos > filp->f_mapping->host->i_size)
        return 0;
    retval = generic_file_direct_IO(READ, iocb, iov, *ppos, 1);
    if (retval > 0)
        *ppos += retval;
    file_accessed(filp);
    return retval;
}
```

函数检查file->f_pos的当前值,文件大小,请求字节数目,接着调用generic_file_direct_IO()函数。当generic_file_direct_IO()函数结束时,__generic_file_aio_read()更新file->f_pos,设置文件的inode的访问时间戳,然后返回。

对设置了O_DIRECT flag了的write()系统调用,文件的write方法最后调用的是generic_file_aio_write_nolock():这个函数检测O_DIRECT flag是否设置了,如果设置了调用generic_file_direct_IO()函数。

generic_file_direct_IO()函数执行步骤:

1. 从kiocb->ki_filp得到file文件对象,地址保存在file变量中,保存file->f_mapping到mapping变量中。
2. 如果操作类型是WRITE并且一个或多个进程创建了一个对文件某部分的内存映射,它调用unmap_mapping_range()解除文件的所有页面映射。这个函数也确保如果要解除映射的页面的对应Page Table entry的Dirty位被设置了,那么在page cache中的对应页面被标记为dirty。
3. 如果它的radix tree不为空(mapping->nopages大于0),它调用filemap_fdatawrite()和filemap_fdatawait()函数刷新所有dirty页面进磁盘,并且等待I/O操作完成。
4. 调用mapping的direct_IO方法。
5. 如果操作类型是WRITE,它调用invalidate_inode_page2()扫描radix tree所有页面并且释放它们。函数也清对应这些页面的空用户态Page Table entries。

16.4. Asynchronous I/O

"Asynchronous"本质上意味着当一个用户态进程调用一个库函数读或写一个文件时,函数在读或写操作入队后就结束了,可能此时真正的I/O数据传输都还未发生。在数据传输进行时,进程可以接着执行。

使用异步I/O是十分简单的。应用程序使用通用的open()系统调用打开文件。接着,填充struct aiocb,这个数据结构中常用的成员有:

1. aio_fildes:文件的file描述符(open函数的返回值)。
2. aio_buf:用户态buffer。
3. aio_nbytes:需要传输的字节数。
4. aio_offset:读或写文件的初始位置。

最后,应用程序把这个aiocb数据结构传递给aio_read()或aio_write();这两个函数一旦请求的I/O数据传输入队就会返回。应用程序之后能调用aio_erro()来检查I/O操作的状态,如果返回EINPROGRESS,说明数据传输还在进行中,如果返回0,说明数据传输已经成功完成,如果数据传输失败,则返回一个错误码。aio_return()函数返回一个已经完成的异步I/O操作有效读或写的字节数目,如果有错误发生则返回-1。

16.4.1. Asynchronous I/O in Linux 2.6

Linux2.6内核实现了一组系统调用用来为异步I/O服务。

16.4.1.1. The asynchronous I/O context

如果用户态进程打算利用io_submit()系统调用开始一个异步I/O操作,那么它必须创建一个asynchronous I/O context。

一个asynchronous I/O context(简称AIO上下文)是一个数据结构集合,用来最踪异步I/O操作的进行。每个AIO上下文与一个kioctx对象相联系,它保存了与AIO上下文相关的所有信息。一个应用能创建多个AIO上下文;一个给定进程的所有kioctx描述符被链入内存描述符(mm_struct)的ioctx_list成员中。

我们将讨论kioctx对象的细节;首先,我们先分析被kioctx对象使用的一个重要数据结构:AIO环。

AIO环是一个进程用户地址空间的内存buffer,它也能被内核态进程访问。AIO环在用户态的开始地址和长度分别保存在kioctx对象的ring_info.mmap_base和ring_info.mmap_size中。组成AIO环的所有page frame的描述符保存在ring_info.ring_pages指向的数组中。

AIO环本质上是一个循环buffer,内核会把异步I/O操作的完成报告写入这个环形buffer。AIO环的第一个字节包含一个头(一个struct aio_ring数据结构);其它字节保存的是io_event数据结构,每个io_event描述了一个异步I/O操作的完成信息。因为AIO环的页面被映射到了进程的用户地址空间,应用程序能直接检测异步I/O操作的进展,这种机制就避免了使用较慢的系统调用方案。

io_setup()系统调用为进程创建一个新的AIO上下文。它有两个参数:异步I/O操作的最大数目,它基本上决定了AIO环的大小。a pointer to a variable that will store a handle to the context;这个handle也是AIO环的基地址。sys_io_setup()服务程序调用do_mmap()来分配一个新的anonymous memory region给包含AIO环的进程,并且创建和初始化一个kioctx对象来描述AIO上下文。

相反的,io_destroy()系统调用移除AIO上下文;它也销毁了包含AIO环的anonymous memory region。系统调用阻塞当前进程直到所有异步I/O操作完成。

16.4.1.2. Submitting the asynchronous I/O operations

为了开始一些异步I/O操作,应用程序调用io_submit()系统调用。它有3个参数:

1. ctx_id:io_setup()返回的handle,标识了AIO上下文。

2. iocbpp:iocb数组的地址,每个成员描述了一个异步I/O操作符。
3. nr:iocbpp指向的数组长度。

sys_io_submit()服务承租执行下列步骤:
1. 验证iocb描述符数组的合法性。
2. 在mm_struct->ioctx_list链表中搜索对应与ctx_id handle的kioctx对象描述符。
3. 对数组中的每个iocb描述符,它执行下列子步骤:
a. 从aio_fildes成员中得到file对象的地址。
b. 分配并初始化一个新的kiocb描述符。
c. 检查AIO环中是否有一个槽来存放I/O操作完成的结果。
d. 根据I/O操作类型来设置kiocb描述符的ki_retry方法。
e. 执行aio_run_iocb()函数,这个函数调用ki_retry方法开使对应异步I/O操作的I/O数据传输。如果ki_retry方法返回值-EIOCBRETRY,说明异步I/O操作已经被提交但没有被满足:aio_run_iocb()函数在一段时间后将再次调用。否则,它调用aio_complete()把异步I/O操作的完成信息保存在AIO环中。

如果异步I/O操作是读请求,对应kiocb描述符的ki_retry方法用aio_pread()来实现。这个函数执行了file对象的aio_read方法,那么根据aio_read方法的返回值更新kiocb描述符的ki_buf和ki_left成员。最后,aio_pread()返回从文件中读取的有效字节数,或者如果函数确定不是所有的请求的字节数都从文件中有效读取,那么返回-EIOCBRETRY。对绝大多数文件系统,file对象的aio_read方法最后会调用_generic_file_aio_read()函数。假设file的O_DIRECT flag被设置,函数结束时调用generic_file_direct_IO()。在这种情况下,__blockdev_direct_IO()函数不会阻塞正在等待I/O进行,数据传输完成的进程;函数会立即返回。因为异步I/O操作仍然在进行,aio_run_iocb()会被再次调用,这次是被aio_wq工作队列的aio内核线程调用。kiocb描述符跟踪I/O数据传输的进行;最后所有的请求数据将被传输并且完成结果会添加到AIO环。

简单来说,如果异步I/O操作是一个写请求,kiocb描述符的ki_retry方法用aio_pwrite()函数来实现。这个函数本质上执行file对象的aio_write方法,接着根据aio_write方法的返回值更新kiocb描述符的ki_buf和ki_left成员。最后,aio_pwrite()返回有效写入文件的字节数,或者如果函数发现不是所有请求的字节都被传输,则返回-EIOCBPETRY。对绝大多数文件系统来说,file对象的aiio_write方法最后会调用generic_file_aio_write_nolock()函数。假设O_DIRECT flag被设置了,这个函数最后会调用generic_file_direct_IO()函数。

Chapter 17. Page Frame Reclaiming

17.1. The Page Frame Reclaiming Algorithm

Linux的一个迷人之处是,在动态内存分配给用户态进程和内核前所做的检测有些粗糙。

例如,没有严格的检测分配某个用户的所有给进程的RAM总数。相似的,也没有检测被内核使用的disk caches和memory caches大小。

这种对内存的粗糙的检测是一种设计选择，它允许内核用最佳的方式来使用可用RAM。当系统负载很低时，RAM大多数分配给disk caches，少量分配给正在运行进程。然而，当系统负载增加时，RAM大多数分配给进程，caches使用的RAM将被缩小，腾出空间给额外的进程。

在之前的章节看到的，memory caches和disck caches会占用越来越多的page frame，但是不会主动释放它们。这种方案是合理的，因为cache系统不知道是否以及何时进程将重用被缓存的数据，因此不能确定cache中的那些应该被释放。而且，由于第9章描述的demand paging mechanism，用户态进程一旦真正访问页面就会通过缺面异常机制，得到page frame；然而，demand paging当页面不再使用时，无法强迫进程释放page frame。

那么，不久后所有的空闲内存将被分配给进程和caches。Linux内核的Page frame reclaiming算法将从用户进程和内核caches中，"偷取"一些page frame还给buddy内存管理系统。

实际上，page frame回收必须在所有空闲内存被用完前执行。否则，内核会很容易陷入内存请求的死锁链，导致系统崩溃。本质上，为了释放一个page frame，内核必须把它的数据写入磁盘；而且，为了完成这个操作，内核需要另一些page frame（例如，为I/O数据传输分配buffer heads）。如果没有空闲page frame存在了，就没有page frame能被释放。

page frame回收算法的一个目标是保存一个最小的空闲内存池，使内核能从"低内存"情况下安全的恢复。

17.1.1. Selecting a Target Page

page frame reclaiming algorithm（PFRA）的目标是收集page frame并且使它们变为空闲。PFRA根据页面内容使用不同的方法处理page frames。我们把page frame分为下面几种类型：unreclaimable pages，swappable pages，syncable pages，和 discardable pages：

Type of pages	Description	Reclaim action
Unreclaimable	1. Free pages (included in buddy system lists) 2. Reserved pages (with PG_reserved flag set) 3. Pages dynamically allocated by the kernel 4. Pages in the Kernel Mode stacks of the processes 5. Temporarily locked pages (with PG_locked flag set) 6. Memory locked pages (in memory regions with VM_LOCKED flag set)	(No reclaiming allowed or needed)

Swappable	1. Anonymous pages in User Mode address spaces 2. Mapped pages of tmpfs filesystem (e.g., pages of IPC shared memory)	Save the page contents in a swap area
Syncable	1. Mapped pages in User Mode address spaces 2. Pages included in the page cache and containing data of disk files 3. Block device buffer pages 4. Pages of some disk caches (e.g., the inode cache)	Synchronize the page with its image on disk, if necessary
Discardable	1. Unused pages included in memory caches (e.g., slab allocator caches) 2. Unused pages of the dentry cache	Nothing to be done

17.1.2. Design of the PFRA

类似与进程调度算法，PFRA也缺乏理论支持，只能依靠经验来设计，因此不能保证PFRA算法不会被修改。但是，设计PFRA的通用的启发式规则应该是保持不变的：

1. 优先释放”无害”的页面：包含在disk和memory caches中，未被任何进程引用的页面，应该比进程的用户地址空间的页面优先回收；回收在caches中的page frame能够不修改任何Page Table entry。在之后的"The Least Recently Used (LRU) Lists"一节中将看到，这条规则稍稍被”swap tendency factor”削弱了。
2. 使所有用户进程的页面变成可回收的：除了被锁住的页面，PFRA必须能够偷取所有属于用户态进程的页面。因此，进程会随着失去它们的page frame而睡眠一个很长的时间。
3. Reclaim a shared page frame by unmapping at once all page table entries that reference it：当PFRA打算释放一个被许多进程共享的page frame，它清空所有引用了该共享page frame的page table entries，并且回收该page frame。
4. 仅回收”unused”页面： PFRA使用一个Least Recently Used (LRU) replacement algorithm来把页面分类为in-use和unused。如果一个页面很长时间没有被访问，那么它在不久后再被访问的概率会很小并且它能被看作”unused”；另一方面，如果一个页面最近被访问过，那么它接下来被访问的概率会很高，它被看作是”in-use”。PFRA仅回收unused页面。

因此，page frame reclaiming algorithm是许多启发式规则的混合体：

Careful selection of the order in which caches are examined.

Ordering of pages based on aging (least recently used pages should be freed before pages accessed recently).

Distinction of pages based on the page state (for example, non-dirty pages are better candidates than dirty pages because they don't have to be written to disk).

17.2. Reverse Mapping

在前面的章节中，PFRA的一个目标是能够释放一个共享的page frame。Linux2.6内核能够快速的定位所有指向同一个page frame的Page Table entries。这个过程被称为反向映射。

一个比较笨的办法是在每个page描述符中增加一个成员，把所有指向page描述符描述的page frame的Page Table entries链接在一起。然而，更新这个链表将显著的增加内核开销；因此，另一种更加高效的解决方案被设计出来了。Linux2.6中所使用的技术被称为object-based reverse mapping。本质上，对任何一个用户态页面，内核在系统中保存了该页面到所属的memory regions的反向链表。每个memory region描述符又保存了对应的memory描述符的指针，最后可以链接到一个Page Global Directory。所以，这些反向链表使PFRA得到了所有引用同一个page的Page Table entries。因为memory region描述符的数量少于page描述符，更新一个共享页面的向后链表消耗的时间会大大减少。让我们看看这种方案是如何执行的。

首先，PFRA必须有办法来判断要回收的页面是共享的还是非共享的，它是mapped还是anonymous的。为了做到这些，内核查看page描述符的两个成员：_mapcount和mapping。

_mapcount成员保存了引用page frame的Page Table entries的数量。计数器从-1开始：这个值意味着没有Page Table entry引用page frame。那么，如果计数器值为0，说明页面是非共享的；如果计数器大于0，说明页面是共享的。page_mapcount()函数接收page描述符地址并且返回它的_maocount成员值加1。

page描述符的mapping成员决定了页面是mapped还是anonymous的。

- 如果mapping成员是NULL，那么页面属于swap cache。
- 如果mapping成员为非NULL并且它的最低bit是1，那么说明页面是anonymous并且mapping成员编码了指向一个anon_vma描述符的指针。
- 如果mapping成员是非NULL并且它的最低bit是0，那么这个页面是mapped；mapping成员指向对应文件的address_space对象。

每个Linux使用的address_space对象都在RAM中，因此它的起始线性地址是4的倍数。所以，mapping的最低bit可以用来作为一个flag，代表了mapping是指向address_space对象的指针还是指向anon_vma描述符的指针。This is a dirty programming trick，但是内核要使用很多page描述符，因此这些数据结构应该尽可能的小。

try_to_unmap()函数接收page描述符的指针，它尝试着清空所有指向page描述符对应的page frame的Page Table entries。函数返回：如果成功的移除所有引用了page frame的Page Table entries，返回SWAP_SUCCESS；如果一些Page Table entries没有被移除，返回SWAP_AGAIN；如果出现了错误，返回SWAP_FALL。函数如下所示：

```
int try_to_unmap(struct page *page)
{
    int ret;
    if (PageAnon(page))
```

```
        ret = try_to_unmap_anon(page);
    else
        ret = try_to_unmap_file(page);
    if (!page_mapped(page))
        ret = SWAP_SUCCESS;
    return ret;
}
```

try_to_unmap_anon()和try_to_unmap_file()函数分别实现了对anonymous pages 和 mapped pages的解除映射。

17.2.1. Reverse Mapping for Anonymous Pages

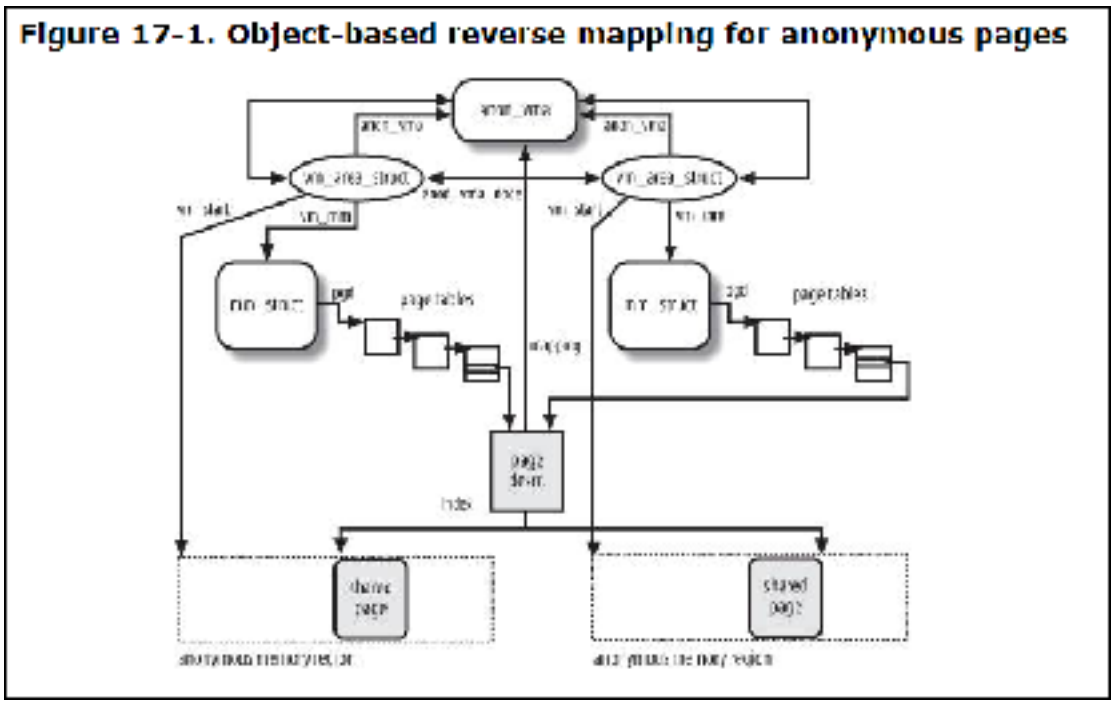
anonymous页面通常被许多进程共享。最常见的情况发生在fork一个新进程。另一种情况发生在当一个进程创建一个memory region,指定MAP_ANONYMOUS和MAP_SHARED flag:这个memory region中的页面将被进程的后代共享。

链接所有引用同一个page frame的anonymous页面的策略十分简单:把包含了该page frame的所有anonymous memory链入一个循环双链表。Be warned that, even if an anonymous memory region includes different pages, there always is just one reverse mapping list for all the page frames in the region.

当内核第一次分配一个page frame给一个anonymous memory,它创建一个新的anon_wma数据结构,它包含了2个成员:lock,一个spin lock来保护链表和head,memory region链表的头。接着,内核把anonymous memory的vm_area_struct描述符插入anon_vma的链表; 相应的,vm_area_struct数据结构包含了两个关于该链表的成员:anon_vma_node,通过它链入anon_vma->head,和anon_vma,它指向了anon_vma数据结构。最后,内核保存了anon_vma数据结构的地址在anonymous page描述符的mapping成员中。

当一个已经被一个进程引用的page frame被插入另一个进程的Page Table entry,内核简单的把第二个进程的anonymous memory region插入第一个进程的mempry reigon的anon_vma->head链表中。所以,anon_vwa中的链表通常包含了不同进程的memory region。

如下图所示,anon_vwa中的链表允许内核快速定位引用了同一个anonymous page frame的所有Page Table entries。实际上,每个memory region描述符的vm_mm保存了memory描述符的地址,memory描述符的pgd成员又包含了进程的Page Global Directory地址。Page Table entry能通过anonymous page的线性地址找到。



17.2.1.1. The try_to_unmap_anon( ) function

当需要回收一个anonymous page frame时，PFRA必须扫描anon_vma的链表中的所有memory regions并且仔细的检查每个region是否真正的包含了该page frame。这个工作是通过try_to_unmap_anon()函数完成的，它接收一个page描述符，本质上执行下列步骤：

1. 对page->mapping指向的anon_vma数据结构，请求它的的anon_vma->lock spin lock。
2. 扫描anon_vma->head链表；把链表中找到的每个memory region描述符保存在vma中，调用try_to_unmap_one()函数。如果try_to_unmap_one()函数返回SWAP_FAIL或，如果page描述符的_mapcount成员表明引用该page frame的所有Page Table entries都被找到了，扫描在到达链表尾前结束。
3. 释放spin lock。
4. 返回最后一次调用try_to_unmap_one()的返回值：SWAP_AGAIN或SWAP_FAIL。

17.2.1.2. The try_to_unmap_one( ) function

try_to_unmap_one()函数被try_to_unmap_anon()进入try_to_unmap_file()两个函数反复的调用。它有两个参数：page和vma。函数本质上执行下列动作：

1. 计算要回收的页面的线性地址，这是通过memory region的起始线性地址(vma->vm_start)，memory region在被映射的文件中的偏移(vma->vm_pgoff)和页面在被映射的文件中的偏移(page->index)。对anonymous页面，vma->vm_pgoff要么是0要么等于vm_start/PAGE_SIZE；对应的，page->index要么是页面在region中的index要么是页面的线性地址除以PAGE_SIZE。
2. 如果目标页面是anonymous，检测页面的线性地址是否落在memory region中；如果没有，返回SWAP_AGAIN并结束(在解释anonymous页面的反向映射时，anon_vma的链表可能包含了没有包含目标页面的memory regions)。
3. 从vma->vm_mm中得到memory描述符的地址，请求vma->vm_mm->page_table_lock spin lock来保护页面表。

4. 连续调用pgd_offset(),pud_offset(),pmd_offset(),和pte_offset_map()来得到目标页面对应的Page Table entry。
5. 执行一些检测来判定目标页面是可回收的。如果下列检测任何一个失败,函数跳转到步骤12结束,返回一个合适的错误码:SWAP_AGAIN或SWAP_FAIL:

a. 检测Page Table entry指向了目标页面;如果没有指向,函数返回SWAP_AGAIN。这种情况发生在:
   - Page Table entry引用的是COW时分配的page frame,但是anonymous memory region仍然在原始page frame的anon_vma->head链中。
   - mremap()系统调用可以重映射 memory region,并且通过直接修改页面表来把页面移动到用户地址空间。在这种特殊情况下,object-based reverse mapping不能工作,因为page描述符的index成员不能用来确定页面的实际线性地址。
   - 文件映射是非线性的。

b. 检查memory region不是locked(VM_LOCKED)或reserved (VM_RESERVED);如果其中一种情况出现,函数返回SWAP_FAIL。
c. 检查Page Table entry的Accessed bit是否被清0了;如果没有,函数清0该位并且返回SWAP_FAIL,因为此时页面被看成是在使用中,那么它不应该被回收。
d. 检查页面是否属于swap cache和它当前是否正在被get_user_pages()处理;在这种情况下,为了避免一个嵌套竞争情况,函数返回SWAP_FAIL。

6. 页面能被回收:如果Page Table entry的Dirty位被设置,设置页面的PG_dirty flag。
7. 清空Page Table entry并且刷新对应的TLB。
8. If the page is anonymous, the function inserts a swapped-out page identifier in the Page Table entry so that further accesses to this page will swap in the page.而且,它减少memory描述符的anon_rss计数器(分配给该进程的anonymous pages数量)。
9. 减少减少memory描述符的rss计数器(分配给该进程的page frame数量)。
10. 递减page->_mapcount,因为引用这个page frame的用户态Page Table entries已经少了一个。
11. 递减page frame的使用计数,它保存在page描述符的_count成员中。如果计数器变成负数,它把page描述符从活跃链表或不活跃链表中移除,并且调用free_hot_page()释放page frame。
12. 调用pte_unmap()释放步骤4中调用pte_offset_map()分配的temporary kernel mapping.
13. 释放vma->vm_mm->page_table_lock spin lock。
14. 返回合适的错误码(如果成功则返回SWAP_AGAIN)。

17.2.2. Reverse Mapping for Mapped Pages

像anonymous那样,mapped页面的object-based reverse mapping基于一个简单的观点: it is always possible to retrieve the Page Table entries that refer to a given page frame by accessing the descriptors of the memory regions that include the corresponding mapped pages.那么,反向映射的核心是一个巧妙的数据结构,它把对应于一个给定page frame的memory region描述符收集起来。

与anonymous页面相反,mapped页面常被共享,因为许多进程可能共享相同的代码页面。例如,考虑标准C库的代码,它几乎被所有系统中的进程共享。因此,Liunx2.6使用了一种

特殊的搜索树,称为"priority search trees",来快速定位引用同一个page frame的所有memory region。

对每一个文件都有一棵优先级搜索树;它的根保存在inode->address_space->i_mmap中。

17.2.2.1. The priority search tree

Linux2.6使用的priority search tree(PST)是基于Edward McCreight在1985年提出的一种数据结构,它用来代表一个overlapping intervals的集合。McCreight的树型结构是一个堆和平衡搜索树的混合体,它用来在intervals集合中执行查询操作。例如:"给定interval中有哪些intervals?"和"哪些intervals与给定interval交叉?",总执行时间正比于树的高度和答案中intervals的个数。

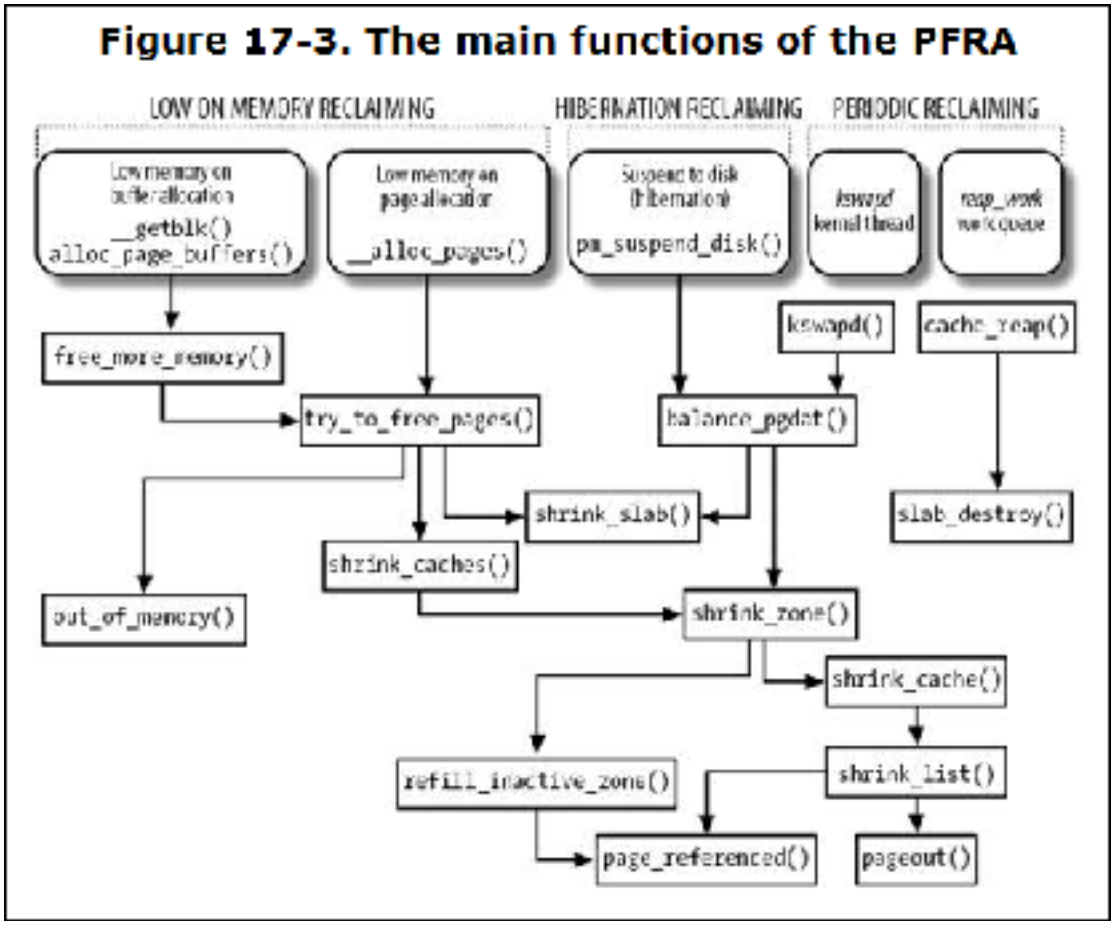
每个PST中的interval对应一个树的节点,它被2个indices描述:the radix index,它对应于interval的起始点,the heap index,它对应于终点。PST本质上是一个关于radix index的搜索数,它带有类似于堆的额外属性:一个节点的heap index大于等于孩子节点的head index。

Linux优先级搜索树与McCreight的数据结构在两个方面有重要的不同:首先,Linux树不是总是保持平衡的(平衡算法在内存空间和执行时间上是代价昂贵的);第二,Linux树合适于存储memory region而不是线性地址intervals。

每个memory region能被看成是一个文件页面的间隔,它可以被在文件中的初始位置(the radix index)和最终位置(the heap index)所标识。而且, memory region倾向于从同一个页面开始(典型的,从页面index0开始)。不幸的是,McCreight的原始数据结构不能保存拥有相同起始点的intervals。作为一个局部解决方案,一个PST的每个节点有一个额外的size index
(暂缺。。。)

17.3. Implementing the PFRA

PFRA必须处理用户态进程,disk cachers和memory caches拥有的各种页面;而且,它必须遵守多种启发试规则。所以,对PFRA由大量函数组成而感到惊奇。图17-3展示了主要的PFRA函数;一个箭头代表了一次函数调用,例如try_to_free_pages()调用了shrink_caches(),shrink_slab()和out_of_memory()。



PFRA有一些"entry points"。实际上,page frame回收算法在三种场合会执行:

1. Low on memory reclaiming:The kernel detects a "low on memory" condition.
2. Hibernation reclaiming:The kernel must free memory because it is entering in the suspend-to-disk state (we don't further discuss this case).
3. Periodic reclaiming:A kernel thread is activated periodically to perform memory reclaiming, if necessary.

Low on memory reclaiming在下列情况下被激活:

- 被__getblk()调用的grow_buffers()函数分配一个新buffer page失败。
- 被creat_empty_buffers()调用的alloc_page_buffers()函数分配临时buffer heads失败。
- __alloc_pages()函数分配一组连续的page frames失败。

Periodic reclaiming被两种不同的内核线程激活:

- kswapd内核线程,它检测某个memory zone中的空闲page frames数量是否已经低于pages_high。
- events内核线程,它预先定义的工作队列的worker threads;PFRA周期性的调度工作队列中的这个线程来回收空闲slabs。

17.3.1. The Least Recently Used (LRU) Lists

所有属于进程用户态地址空间或page cach的页面被组织成两个链表:活跃链表和不活跃链表;它们都是使用LRU链表表示。活跃链表倾向于包含最近被访问的页面,不活跃链表倾向于包含一段时间内未被访问的页面。

活跃链表和不活跃链表是页面回收算法的核心数据结构。这两个链表的头部分别保存在zone描述符的active_list和inactive_list成员中。zone描述符的nr_active 和nr_inactive成员保存了这两个链表中的页面数目。最后,lru_lock成员是一个spin lock,可以保护两个链表在SMP系统中不被同时访问。

如果一个页面属于一个LRU链表,它的页面描述符中的PG_lru flag被设置。而且,如果页面属于活跃链表,PG_active flag被设置,如果属于不活跃链表,PG_active flag被清0。通过页面描述符的lru成员链入这两个链表。

lru_cache_add()和lru_cache_add_active()这两个函数稍微有些复杂。实际上,这两个函数并不是立刻把页面移动到LRU中;它们把页面累积在数据结构pagevec中,数据结构包含了14个page描述符指针。当一个pagevec数据结构完全被填满时,页面才会被移动到LRU链中。这中机制增强了系统性能,因为请求LRU spin lock的次数减少了。

17.3.1.1. Moving pages across the LRU lists

PFRA收集了活跃链表中最近被访问的页面,使之在选择可被回收的page frames时不再被扫描。相反的
(暂缺。。。)

17.3.1.2. The mark_page_accessed( ) function

当内核必须标记页面为"被访问过",它调用mark_page_accessed()函数。这个事件发生在内核发现页面被一个用户态进程,一个文件系统层,或一个设备驱动引用过。例如,mark_page_accessed()在如下情况下被调用:

- When loading on demand an anonymous page of a process (performed by the do_anonymous_page() function; see the section "Demand Paging" in Chapter 9).
- When loading on demand a page of a memory mapped file (performed by the filemap_nopage() function; see the section "Demand Paging for Memory Mapping" in Chapter 16).
- When loading on demand a page of an IPC shared memory region (performed by the shmem_nopage() function; see the section "IPC Shared Memory" in Chapter 19).
- When reading a page of data from a file (performed by the do_generic_file_read() function; see the section "Reading from a File" in Chapter 16).
- When swapping in a page (performed by the do_swap_page() function; see the section "Swapping in Pages" later in this chapter).
- When looking up a buffer page in the page cache (see the __find_get_block() function in the section "Searching Blocks in the Page Cache" in Chapter 15).

如果这个函数在被调用之前PG_referenced flag被设置了,那么函数会把页面从不活跃链表移动到活跃链表。

17.3.1.3. The page_referenced( ) function

对每个被PFRA扫描过的页面都会调用一次page_referenced()函数,如果PG_reference flag或引用该page frame的Page Table entries中有一个Accessed位被设置了,函数返回1;否则返回0。这个函数首先检测页面描述符的PG_referenced falg;如果flag设置了,它清0该位。接着,函数利用了object-based reverse mapping机制来检测并清0 Accessed位。为了做到这一点,函数利用了3个辅助函数;page_referenced_anon(), page_referenced_file(), 和

page_referenced_one()，他们和try_to_unmap_xxx()函数很相似。The page_referenced( ) function also honors the swap token; see the section "The Swap Token" later in this chapter.

page_referenced()函数不会把页面从活跃链表移动到非活跃链表；这个工作是被refill_inactive_zone()函数完成的。

17.3.1.4. The refill_inactive_zone( ) function

refill_inactive_zone()函数是被shrink_zone()调用，shrink_zone()函数执行对page cache和用户地址空间的页面的回收。函数接收2个参数：指针zone指向一个memory zone描述符，指针sc指向一个scan_control结构。scan_control数据结构被PFRA广泛的使用，并且包含了关于回收操作过程的信息。

refill_inactive_zone()函数的角色是很重要的，因为它把页面从一个活跃链表移动到一个不活跃链表。如果这个函数执行力度太大，它会从活跃链表中移动过多的页面到不活跃链表中；这会导致PFRA回收大量的page frames，并且系统性能将被降低。另一方面，如果函数执行力度太小，非活跃链表不能被足够的未使用页面补充，PFRA在回收内存时将失败。于是，函数实现了一种adaptive behavior：它每次调用时，在不活跃链表中扫描少量的页面；如果PFRA在回收page frames时碰到了麻烦，refill_inactive_zone()函数增加每次调用时扫描的页面数。这个行为被scan_control->priority成员控制。（值越小优先级越高）。

另一个启发式规则控制着refill_inactive_zone()函数的行为。LRU链表包含了2种页面：属于用户地址空间的页面和包含在page cache中的不属于任何用户进程的页面。（暂缺）

17.4. Swapping

Swapping为非映射页面提供了一个备份。我们知道有3种页面必须被swapping子系统处理：

- 属于一个进程的anonymous memory region页面（用户栈和堆）
- 属于进程的私有内存映射的dirty页面
- 属于一个IPC共享memory region的页面

类似于demand paging，swapping必须对程序透明。换句话说，没有相关与swpping的特殊指令需要被插入代码。为了理解这个是如何实现的，回忆章节"Regular Paging"，每个Page Table entry包含了一个Present flag。内核利用这个flag来表明属于进程地址空间的页面已经被swapped out。算上该flag，Linux还利用Page Table entry的剩下的位来保存一个”swapped-out page identifier”，这个标识符编码了磁盘上的swapped-out page的位置。当一个Page Fault异常发生时，对应的异常处理程序检测到该页面不在RAM中，并且调用函数从磁盘中把缺失的页面swap进RAM。

Swapping 子系统的主要特征被总结如下：

- 在磁盘上建立”swap areas”来保存还没有磁盘镜像的页面。
- 管理swap areas上的空间，在需要时的分配和释放”page slots”，
- 提供函数，从RAM中"swap out"页面进一个swap area和从磁盘中”swap in”页面进RAM。

- 利用在页面的Page Table entries中的”swapped-out page identifier”，来追踪数据在swap areas中的位置。

总之，swapping是页面回收的最重要的特性。如果我们希望确保进程拥有的所有的page frames，而不仅是那些在磁盘上有镜像的页面(*我觉的是只映射文件*)会被PFRA回收，那么swapping必须被使用。当然，你可以通过swappoff命令关闭swapping；这种情况下，当系统负载增加时，disk thrashing很有可能在不久后发生。

我们应该注意到，swapping能被用来扩展用户态进程有效使用的内存地址空间。事实上，大的swap areas允许内核加载许多所需内存超过物理RAM总量的应用。一个进程访问一个当前被swapped out的页面消耗的时间比访问在RAM中页面的消耗的时间多出多个数量级。简而言之，如果性能是非常重要的，swapping应该仅被用做是最后的手段；增加RAM是应对计算量不断增加的最好的好解决方案。

17.4.1. Swap Area

页面从内存中swapped out，然后被保存进一个swap area，swap area的实现是通过一个磁盘分区或一个大磁盘分区中的一个文件。可以定义多个不同的swap area，最多MAX_SWAPFILES(通常设置为0)。

拥有多个swap areas允许系统管理者在多个磁盘中使用swap空间，使硬件能同时执行；也能让swap空间在运行时被增加，而不用重启系统。

每个swap cache由一组连续的page slots组成：4096字节的块，用来包含一个swapped-out页面。Swap area的第一个page slot被用来永久的保存关于该swap area的信息；它的格式被swap_header描述，这个union由两个数据结构组成：info和magic。magic数据结构包含了一个字符串，明确的表明磁盘的这个部分是一个swap area。magic数据结构本质上是允许内核明确的标识一个文件或一个分区为swap area；字符串的文本为”SWAPSPACE2”，总是保存在第一个page slot的末尾。

info数据结构包含下列成员：

- bootbits

Not used by the swapping algorithm; this field corresponds to the first 1,024 bytes of the swap area, which may store partition data, disk labels, and so on.

- version

Swapping algorithm version.

- last_page

Last page slot that is effectively usable.

- nr_badpages

Number of defective page slots.

- padding[125]

Padding bytes.

- badpages[1]

Up to 637 numbers specifying the location of defective page slots.

17.4.1.1. Creating and activating a swap area

存储在swap area中的数据只在系统运行时有效。当系统被关闭，所有进程被杀死，保存在swap area中的数据将被丢弃。因此，swap area包含了非常少的控制信息：swap area type和缺陷page slots链表。这些控制信息很容易就能放在一个单一的4KB页面中。

通常,系统管理员在Linux系统上创建其它分区时会创建一个swap分区,然后使用mkswap命令把一个磁盘分区设为swap area。这个命令初始化了第一个page slot。因为磁盘可能包含了一些坏块,程序也会检测所有page slots来定位有缺陷的slot。执行maswap会使swap area处于一个非激活状态。每个swap area会在系统启动或系统运行时动态激活。

每个swap area由一个或多个swap extents组成,每个swap extents由一个swap_extent描述符代表。每个extent对应一组在磁盘物理上邻近的page slots。swap_extent描述符包含了swap area中extent的第一个页面,extent的页面数量,extent的起始磁盘sector号。一个extents的有序链表组成了一个swap area。储存在一个磁盘分区上的swap area只由一个extent组成;相反的,储存在一个常规文件上的swap area由多个extents组成,因为文件系统可能没有分配连续的磁盘块给整个文件。

17.4.1.2. How to distribute pages in the swap areas

当swap out一个页面,内核尝试把页面保存在连续的page slots中,这样可以最小化磁盘寻道时间;这是一个高效swapping算法的重要元素。

然而,如果有超过一个的swap area被使用,事情变得更加复杂。保存在较快的磁盘上的较快的swap areas有更高的优先级。当内核查找一个空闲slot时,搜索从拥有最高优先级的swap area开始。如果有多个相同优先级的swap area,则是通过循环选择来避免某一个负载过高。

17.4.2. Swap Area Descriptor

每个在内存中活跃swap area有自己的swap_info_struct描述符。

flags成员包含了3个重叠子成员:

- SWP_USED:如果swap area为活跃,该位为1;否则为0。
- SWP_WRITEOK:如果可以往swap area中写进数据该位为1;如果swap area是只读的该位为0(它正在被激活或处于非活跃状态)。
- SWP_ACTIVE: SWP_USED和SWP_WRITEOK的组合;当SWP_USED和SWP_WRITEOK都被设置时,该成员被设置。

swap_map成员指向一个计数器数组,每个元素对应一个swap area page slot。如果计数器等于0,page slot为空闲;如果它是正数,page slot被一个swapped-out页面填充了。本质上,page slot计数器代表了共享该swapped-out页面的进程数量。如果计数器值为SWAP_MAP_MAX(等于32767),保存在page slot中的页面是"永久的",它不能从相应的slot中移除。如果计数器的值为SWAP_MAX_BAD(等于32768),page slot被看成是有缺陷的,那么它不能被使用。

prio成员是一个有符号整数,代表了the order in which the swap subsystem should consider each swap area.

swap_info全局数组报了MAX_SWAPFILES个swap area描述符。仅当它的SWP_USED falgs被设置了,该swap area才被使用。

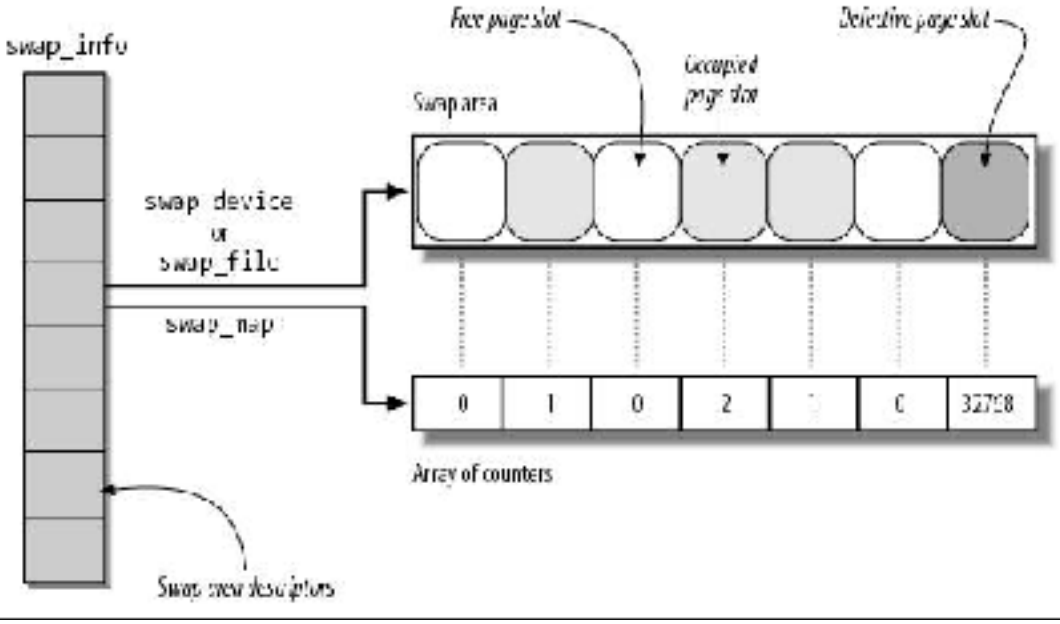
nr_swapfiles全局变量保存了swap_info数组中最后一个有效元素的index。注意这个变量的名字,但它并不代表活跃swap area的数量。

活跃swap areas的描述符被插入一个用优先级排序的链表中。链表是通过swap areas描述符的next成员来实现的,它保存了swap_info数组中下一个描述符的index。

swap_list_t类型的swap_list全局变量,包含了下列成员:

- head:swap_info数组中第一个链表元素的index。
- next: Index in the swap_info array of the descriptor of the next swap area to be selected for swapping out pages. This field is used to implement a Round Robin algorithm among maximum-priority swap areas with free slots.

Figure 17-6. Swap area data structures



swap area描述符的max成员保存了swap area的大小(以页面数计算),pages成员保存了可用page slot数量。这两个成员值是不同的,因为pages没有考虑第一个page slot和有缺陷的page slots。

最后,nr_swap_pages全局变量包含了所有活跃swap areas中的可用page slots总数(空闲的和没有缺陷的page slots)。

17.4.3. Swapped-Out Page Identifier

一个swapped-out页面可以方便的用一个swap_info数组中的index和一个swap area中的page slot index来唯一的标识。因为swap area中的第一个page slot(index为0)被保留用作swap_header union,因此第一个可用的page slot的index为1.一个swapped-out page identifier的格式如下图所示:

Figure 17-7. Swapped-out page identifier

Swp_entry(type,offset)函数使用swap area index type和page slot index offset构造了一个swapped-out page identifier。相反地,swap_type和swp_offset函数从一个swapped-out page identifier中分别抽取swap area index和page slot index。

当一个页面被swapped out，它的identifier别写入对应的Page Table entry。注意到entry的最低位是Present flag,此时该位总为0,代表着该页面现在不在RAM中。然而,剩下的31位至少有一位是1,因为页面不可能被交swap到0号swap area的0号page slot。因此,Page Table entry能代表3中不同情况:

- Null entry:页面不属于该进程的地址空间,或者是该page frame还没有被分配给该进程(demand paging)。
- First 31 most-significant bits not all equal to 0, last bit equal to 0:页面已被swap out.
- Least-significant bit equal to 1:页面在RAM中。

在80X86体系中,24bits限制了一个swap area大小为2^24slots(64GB)。

因为一个页面可能属于多个进程地址空间,页面可能从一个进程地址空间被swap out却仍然存在于主存中;因此,swap out同一个页面许多次是可能的。一个页面只会被被物理的swap out一次,以后每次swap该页面只会增加swap_map计数器。

swap_duplicate()函数通常在试图swap out一个已经被swapped-out页面时调用。它简单的验证作为参数传进来的swapped-out page identifier的合法性并且递增相应的swap_map计数器。

17.4.4. Activating and Deactivating a Swap Area

一旦一个swap area被初始化了,超级用户可以使用swapon和swapoff分别来激活和解除激活swap area。

17.4.4.1. The sys_swapon( ) service routine

sys_swapon()服务程序接收下列参数:

- specialfile:这个参数指向用来实现swap area的设备文件或常规文件的路径名。
- swap_flags:这个参数由一个单一的SWAP_FLAG_PREFER位加上31bits的swap area优先级(仅当SWAP_FLAG_PREFER置位时,优先级才有效)组成。

函数检查swap_area中第一个slot中的swap_header成员。函数主要执行下列步骤:

1. 检测当前进程是否有CAP_SYS_ADMIN权限。
2. 在swap_info数组中循环查找第一个SWP_USED flag清0的swap area描述符,这意味着对应的swap area是非活跃的。循环次数不能超过nr_swapfiles。如果一个非活跃swap area被找到,跳转到步骤4。
3. 检测添加一个新的swap area是否可行;如果不能添加了,返回一个错误码;否则,把nr_swapfiles 递增1。
4. 一个未使用的swap area的index被找到了:它初始化描述符的成员;特别的,设置flags为SWP_USED,并且设置lowest_bit和highest_bit为0。
5. 如果swap_flags参数指定了新swap area的优先级,函数设置描述符的prio成员。否则,初始化该成员为所有活跃swap area中最低优先级减一。如果没有其它活跃的swap areas是活跃的,函数设置它为-1。

6. 从用户空间拷贝specialfile指向的字符串。
7. 调用filp_open()来打开specialfile指定的文件。
8. 把filp_open()返回的file对象地址保存在swap area描述符的swap_file成员中。
9. 确认这个swap area之前没有被激活。这是通过检测swap_file->f_mapping成员来确定的。如果swap area已经被激活了,返回一个错误码。
10. 如果specialfile参数指向的是一个块文件,它执行下列子步骤:
a. 调用bd_claim()设置swapping子系统为块设备的holder。如果块设备已经有一个holder,它返回一个错误码。
b. 保存block_device描述符地址到swap area描述符的bdev成员中。
c. 保存设备的当前块大小到swap area描述符的old_block_size成员中,接着设置设备的块大小为4096字节(页面大小)。
11. 如果specialfile参数指向的是一个常规文件,它执行下列子步骤:
a. 检测文件的inode->i_flags成员的S_SWAPFILE成员。如果该flag被设置了,返回一个错误码,因为文件已经被用作一个swap area。
b. 保存包含该文件的块设备的描述符地址到swap area描述符的bdev成员中。
12. 读取保存在swap area的slot 0中的swap_header描述符。为了这个目的,调用read_cache_page()。等待该页面被读进内存。
13. 检测第一个页面的最后10个字符是否等于"SWAPSPACE2"。如果不,返回一个错误码。
14. 根据swap_header->info.last_page中的swap area大小,初始化swap area描述符的lowest_bit和highest_bit成员。
15. 调用vmalloc()创建与新swap area相关的计数器数组并且把它的地址保存在swap描述符的swap_map成员中。根据保存在swap_header->info.bad_page中的缺陷page slots链表,初始化数组元素为0或SWAP_MAP_BAD。
16. 访问info.last_page和info.nr_badpages成员计算出可用page slots数量,并把结果保存在swap area描述符的pages成员中。设置max成员为swap page中的页面总数。
17. 为新swap area(如果swap area是磁盘分区,只创建一个swap extent)创建swap extents链表extent_list,并且设置合适的swap area描述符的nr_extents和curr_swap_extent成员。
18. 设置swap area描述符的flags为SWP_ACTIVE。
19. 更新全局变量nr_good_pages, nr_swap_pages, and total_swap_pages。
20. 把swap area描述符插入swap_list变量指向的链表。
21. 返回0(成功)。

17.4.4.2. The sys_swapoff( ) service routine

sys_swapoff()服务程序解除激活一个swap area。它比sys_swapon()更复杂也更耗时,因为要解除激活的分区可能包含属于许多进程的页面。函数扫描swap area并且强迫其中的页面swap进RAM。因为swap-in需要一个新的page frame,如果没有空闲的page frame就会失败。在这种情况下,函数返回一个错误码。函数主要执行下列步骤:

1. 检测当前进程是否有CAP_SYS_ADMIN权限。
2. 拷贝specialfile参数指向的字符串进内核空间。
3. 调用fil_open()打开文件;函数返回file对象的地址。
4. 扫描swap area描述符的swap_list链表,查看fil_open()返回的file描述符是否在链表中。如果不在,说明传递给函数的参数是无效的,返回一个错误码。

5. 调用cap_vm_enough_memory()检查是否有足够的空闲page frame来保存swap area中的页面。这种检测是很粗糙的,但是它避免了内核做大量无用的磁盘活动。cap_vm_enough_memory()统计了通过slab caches分配的并且SLAB_RECLAIM_ACCOUNT flag置位的page frames。这种页面的总数,保存在slab_reclaim_pages变量中。
6. 把swap area描述符从swap_list链表中移除。
7. 更新nr_swap_pages和total_swap_pages变量。
8. 清0该swap area描述符的flags成员的SWP_WRITEOK flag;这将阻止PFRA把更多页面交换进swap area。
9. 调用try_to_unuse()把swap area中的所有页面交换进RAM并且相应的更新使用这些页面的Page Tables。当执行这个函数时,当前进程,也就是执行了swapoff命令的进程,它的PF_SWAPOFF flag置位。设置PF_SWAPOFF flag的后果是:万一发生了page frames的短缺,select_bad_process()函数将杀死该进程。
10. 等待包含该swap area的块设备驱动被upplugged。在这种情况下, 在swap area被解除激活前,被try_to_unuse()调用的读请求将被驱动处理。
11. 如果try_to_unuse()不能够分配所有请求的page frames,swap area不能被解除激活。所以,函数执行下列子步骤:

a. 把swap area描述符重新插入swap_list链表并且设置flags成员为SWP_WRITEOK。
b. 恢复nr_swap_pages和total_swap_pages变量的初始值。
c. 调用filp_close()关闭步骤3打开的文件并且返回一个错误码。

12. 否则,所有被使用的page slots已经被成功的传输进RAM。所以,函数执行下列子步骤:

a. 释放被swap_map数组的内存和extent描述符。
b. 如果swap area是在磁盘分区中的,函数从swap area描述符的old_block_size成员,恢复磁盘分区的原始块大小 ;而且,调用bd_release()函数使swap子系统不再持有该块设备。
c. 如果swap area被保存在一个常规文件中,清0文件的inode的S_SWAPFILE flag。
d. 调用filp_close()两次,第一次是关闭swap_file file对象,第二次关闭步骤三打开的文件。
e. 返回0(成功)。

17.4.4.3. The try_to_unuse( ) function

try_to_unuse()函数把页面从swap area中换进内存并且更新引用了该页面的进程的Page Tables。为了这个目的,函数访问了所有内核线程和用户进程的地址空间,这是从init_mm这个memory描述符开始,循环的遍历memory描述符链表。这个过程是很耗时的,在执行过程中中断是打开的。多个进程的同步是很关键的。

try_to_unuse()函数扫描了swap area的swap_map数组。当函数找到一个使用中的page slot,它首先把该页面交换进内存,然后开始查找引用了该页面的进程。这两个操作的顺序是很重要的,它避免的竞争条件。当I/O数据传输在进行时,页面被锁定了,因此没有进程能访问它。一旦I/O数据传输完成,页面再次被try_to_unuse()函数锁定,因此页面不能被另一个内核控制流交换出去。因为每个进程在开始一个swap-in或一个swap-out操作时会查找page cache,竞争条件也会被避免。最后,swap area被try_to_unuse()函数标记为不可写(SWP_WRITEOK flag被设置),因此没有进程能在这个swap area上执行一个swap-out操作。

However, try_to_unuse() might be forced to scan the swap_map array of usage counters of the swap area several times. This is because memory regions that contain references to swapped-out pages might disappear during one scan and later reappear in the process lists.

例如，回忆do_munmap()函数：当一个进程释放一个线性地址interval时，do_munmap()把进程中所有包含了受影响线性地址的memory regions移除；不久后，函数再把部分被解除映射的memory region插入进程。do_munmap()函数只会把被释放线性地址interval对应的swapped-out页面释放掉。

try_to_unuse()函数在寻找一个引用了给定page slot的进程会失败，因为对应的memory region只是暂时的没有包含在进程的链表中。为了应对这种情况，try_to_unuse()持续的扫描swap_map数组，直到是所有引用计数变成0。最后当引用了该页面的memory region重新出现在进程的链表中时，try_to_unuse()函数就能成功的释放所有page slots。

现在让我们来描述try_to_unuse()的主要执行过程。它执行一个连续的循环，循环次数为swap_map数组中的引用计数器。循环是可中断的并且当函数收到一个信号时会返回一个错误码。在每次循环中，函数执行下列步骤：

1. 如果计数器等于0(没有页面保存在这了)或等于SWP_MAP_BAD，那么对下一个page slot进行操作。
2. 否则，它调用read_swap_cache_async()函数把页面交换进内存。这个函数在必要时会分配一个新page frame用page slot中的数据填充它，然后把页面放入swap cache中。
3. 等待直到新页面被更新然后锁定它。
4. 当函数在执行前一个步骤时，进程会被挂起。所以，它再次检测page slot的使用计数器是否为0；如果是，这个swap page已经被另一个内核控制流释放了，因此函数接着处理下一个page slot。
5. 针对从init_mm开始的链表中的每个memory region，调用unuse_process()。这个耗时函数扫描了所有进程的Page Tables，并更新。函数也会递减page slot的计数器和递增page frame的使用计数器。
6. 调用shmem_unuse()检测swapped-out page是否被用做一个IPC共享内存资源并且恰当的处理这种情况。
7. 检测页面的引用计数值，如果等于SWAP_MAP_MAX，page slot是”永久的”。为了释放它，把计数器置为1。
8. swap cache可能也拥有这个页面。如果页面属于swap cache，它调用swap_writepage()函数把它的内容刷新进磁盘(如果页面是dirty)并且四奥用delete_from_swap_cache()把页面从swap cache中移走，递减它的引用计数。
9. 设置page描述符的PG_dirty flag，解锁page frame，递减引用计数。
10. 检测当前进程的need_resched成员；如果该位设置了，调用schedule()释放CPU。解除激活一个swap page是一个长时间的工作，内核必须确保系统中的其它进程持续执行。当该进程再次被选择运行时，try_to_unuse()函数将接着执行。
11. 处理下一个page slot，从步骤1开始。

17.4.5. Allocating and Releasing a Page Slot

当释放内存时，内核在段时间内会把许多页面swap out。因此，把这些页面保存在连续的slots中可以最小化磁盘寻道时间。

Chapter 18. The Ext2 and Ext3 Filesystems

18.1. General Characteristics of Ext2

下列特性有利于提高EXT2文件系统的性能：

- 当创建一个EXR2文件系统，系统管理员需要选择最优的块大小（从1024到4096字节），这个选择依赖于期望的平均文件大小。例如，如果平均文件大小少于数千字节，那么1024字节的块大小是最优的，因为这将使内部碎片比较少。从另一个角度，大的块大小通常比较性能高，因为它能减少磁盘传输。
- 当创建一个EXT2文件系统时，系统管理员可以选择inodes的数量，这取决于期望保存的文件数量。
- 文件系统把磁盘块组织成组。每个组包含数据块和inodes，它们都是物理上邻近的。保存在一个数据块组中的文件被访问时会有较小的磁盘寻道时间。
- 文件系统会在常规文件真正使用前，预分配一些数据块给它。那么，在文件大小增加时，一些块已经保存在物理上邻接的位置，这将减少了文件碎片。
- 支持快速符号链接。如果符号链接代表了一个短的路径名（最多60字节），它将保存在inode中并能不读取磁盘块而被转换。

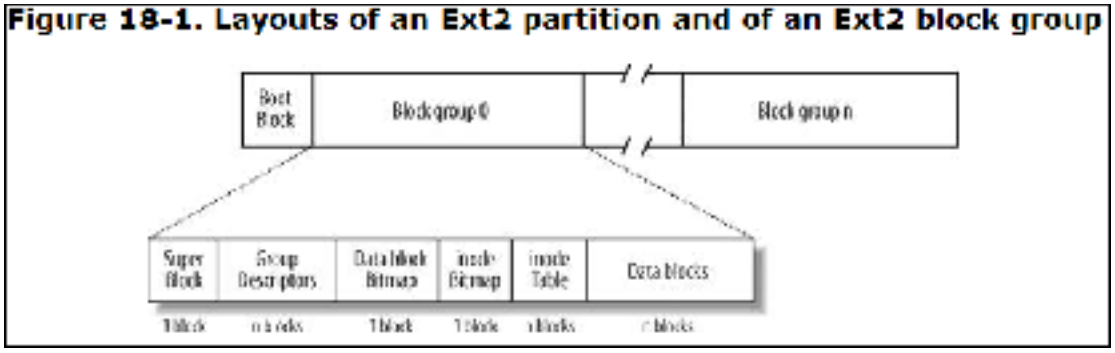
而且，EXT2文件系统包括了其它特性来使它健壮和灵活：

- 一个仔细实现的文件更新过程，最小化了系统崩溃的影响。例如，当创建一个新的指向一个文件的硬链接，在磁盘inode中的硬链接的计数器首先加一，然后新的名字加在合适的目录中。使用这种方法，如果硬件故障发生在目录更新之前，那么目录仍然是连续的，即使inode的硬链接计数器是错的。虽然文件的数据块不能自动回收，但删除文件也不会导致灾难性的结果。
- 支持在启动时自动检测文件系统的连续性。
- 支持不可变文件（不能被修改，删除或改名）和仅追能加文件（数据仅能被添加到文件尾部）。

18.2. Ext2 Disk Data Structures

在每个Ext2分区中的第一个块并不被Ext2文件系统管理，因为它被留作该分区的启动扇区。Ext2分区的其它部分被划分成块组，每个块组的布局如下图所示。所有的块组在文件系统中有同样的大小并被连续的储存，这样内核能简单的通过整数索引来定位块组的位置。

Figure 18-1. Layouts of an Ext2 partition and of an Ext2 block group



块组减少了文件碎片，因为内核如果可能，会尝试保持属于文件的数据块在同一个块组中。块组中的每个块包含了下列信息之一：

- 文件系统的超级块的一个拷贝。
- 块组描述符组的拷贝。
- 一个数据块位图。
- 一个inode位图。
- 一个inodes表。
- 属于一个文件的数据块。

18.2.1. Superblock

一个Ext2磁盘超级块保存在一个ext2_super_block结构中。

s_inodes_count成员储存了inode数量，s_blocks_count成员储存了块的数量。

s_log_block_size成员代表了块大小，以1024为单位。例如，0代表1024字节的块，1代表2046字节的块。s_log_frag_size成员目前等于s_log_block_size，因为块碎片还未被实现。

s_blocks_per_group, s_frags_per_group, 和 s_inodes_per_group成员分别储存了每个块组中块数量，碎片数量，inode数量。

一些磁盘块被留给超级用户（或留给一些其他用户或组，这是通过s_def_resuid和s_def_resgid成员来指定的）。这些快允许系统管理者持续使用文件系统，即使对普通用户来说已经没有空闲磁盘块了。
s_mnt_count, s_max_mnt_count, s_lastcheck, 和 s_checkinterval被用于e2fsck来检测文件系统。

18.2.2. Group Descriptor and Bitmap

18.2.2. Group Descriptor and Bitmap

每个块组有自己的组描述符：ext2_group_desc结构。

当分配新的inodes和数据块时，bg_free_blocks_count,bg_free_inodes_count和bg_used_dirs_count将被使用。

18.2.3. Inode Table

inode table包含了预先定义数目的inode。 inode table所在的初始磁盘块号保存在组描述符的bg_inode_table中。

所有的inode都是同样的大小：128字节。

每个Ext2 inode是一个ext_inode结构。

i_size成员保存了文件的有效长度，以字节计算，i_block成员保存了分配给文件的数据块数量。

i_size和i_block并不一定相关。因为一个文件总是保存在整数个块中，一个非空的文件至少占一个数据块。一个文件可能含有空洞，在这种情况下，i_size可能大于512X i_blocks。

i_block成员是一个EXT2_N_BLOCKS（通常是15）大小的块指针数组，指向了分配给该文件的数据块。

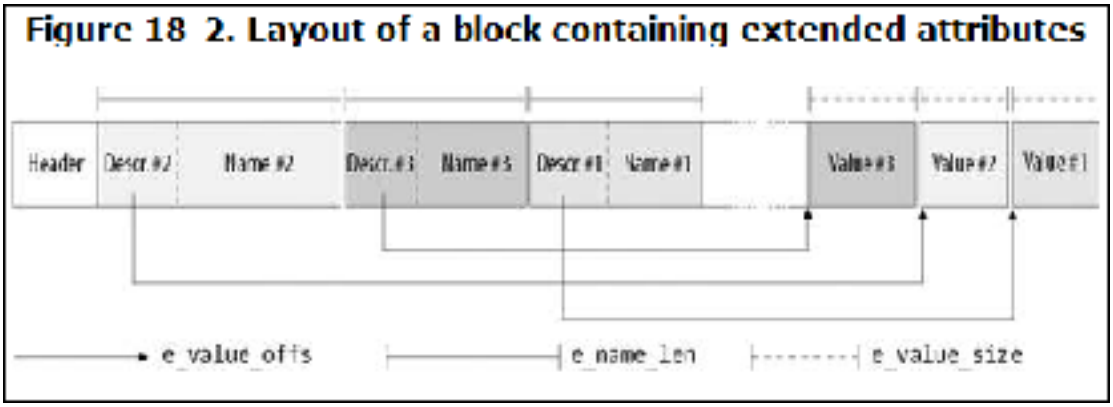
18.2.4. Extended Attributes of an Inode

Ext2 inode格式是对文件系统设计者的一种束缚。一个inode的长度必须是2的幂，这样就能避免在保存inode table的块中出现内部碎片。实际上，一个Ext2 inode的128字节目前已经装载了很多信息，几乎没有空间留给额外的成员。从另一个角度考虑，把inode长度扩充到256字节是十分浪费的，而且会引发兼容性问题。

Extended attributes被引入来克服上述限制。这些属性被保存在另外的磁盘块中。inode->i_file_acl成员指向一个包含了extended attributes的块。拥有同一个extended attributes集合的不同inode共享该磁盘块。

每个extended attribute有一个名字和一个值。它们都被编码进了一个可变长度的字符数组。每个属性被分割成两个部分：ext2_xattr_entry描述符把属性名放在块的开始，同时把属性值放在块的末尾。

Figure 18-2. Layout of a block containing extended attributes



18.2.6. How Various File Types Use Disk Blocks

Table 18-4. Ext2 file types

File_type	Description
0	Unknown
1	Regular file
2	Directory
3	Character device
4	Block device
5	Named pipe
6	Socket
7	Symbolic link

18.2.6.2. Directory

Ext2把目录实现为一种特殊的文件，它的数据块储存了文件名和对应的inode号。这种数据块包含了类型为ext2_dir_entry_2的结构。

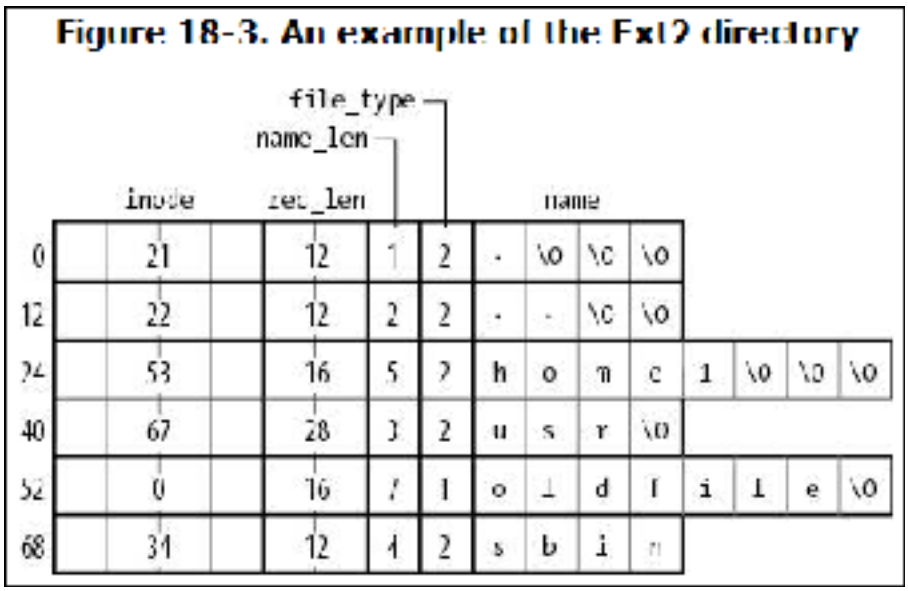


Figure 18-3. An example of the Ext2 directory

18.3. Ext2 Memory Data Structures

Table 18-6. VFS images of Ext2 data structures

Type	Disk data structure	Memory data structure	Caching mode
Superblock	ext2_super_block	ext2_sb_info	Always cached
Group descriptor	ext2_group_desc	ext2_group_desc	Always cached
Block bitmap	Bit array in block	Bit array in buffer	Dynamic
inode bitmap	Bit array in block	Bit array in buffer	Dynamic
inode	ext2_inode	ext2_inode_info	Dynamic
Data block	Array of bytes	VFS buffer	Dynamic
Free inode	ext2_inode	None	Never
Free block	Array of bytes	None	Never

18.3.1. The Ext2 Superblock Object

在Ext2中，VFS超级块的s_fs_info成员指向的是ext2_sb_info结构，它包含了下列信息：

- 绝大多数磁盘超级块成员。
- 一个s_sbh指针，指向包含磁盘超级块的buffer的buffer head。
- 一个s_es指针指向包含磁盘超级块的buffer。
- 能被包含在一个磁盘块中的组描述符数量，s_desc_per_block。
- 一个s_group_desc指针，指向一个包含了组描述符的buffer的buffer head数组。
- 其它关于mount state, mount options等等的数据。

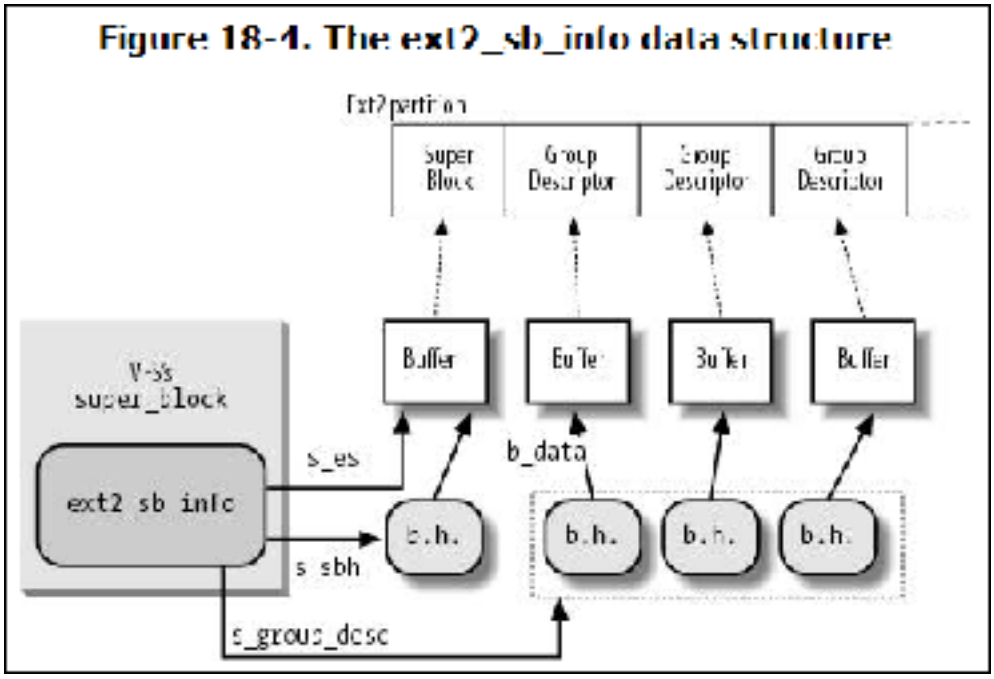


Figure 18-4. The ext2_sb_info data structure

当内核加载一个Ext2文件系统时，它调用ext2_fill_super()函数为数据结构分配空间并从磁盘中读取数据来填充它。下面是该函数的一个简单的描述：

1. 分配一个ext2_sb_info描述符并且把它的地址存放在作为参数传进来的超级块对象的s_fs_info成员中。
2. 调用__bread()来分配一个buffer以及对应的buffer head，并从磁盘块中读取超级块到这个buffer中。把该buffer head地址保存进Ext2超级块对象的s_sbh成员中。
3. 分配一个字节数组，每个字节对应一个组并且保持它的地址到ext2_sb_info->s_debts中。
4. 分配一个指向buffer heads的指针数组，每个buffer heads对应于一个组描述符，把该数组的地址保存到ext2_sb_info->s_group_desc中。
5. 重复调用__bread()来分配buffers并从磁盘中读取包含了Ext2组描述符的块；把buffer head的地址保存到之前分配的s_group_desc数组中。
6. 为root目录分配一个inode和一个dentry对象，在超级块对象中建立一些成员，使随后可以从磁盘中读取root inode。

18.3.2. The Ext2 inode Object

当VFS访问一个Ext2磁盘inode时，它创建一个对应inode描述符，类型为ext2_inode_info。这个描述符包含了下列信息：

- 保存在vfs_inode成员中的，整个VFS inode对象。
- 绝大多数存在于磁盘的inode结构中，但不在VFS inode中的成员。
- 保存inode所属的组号在i_block_group中。
- i_next_alloc_block 和 i_next_alloc_goal成员，保存了最近分配给文件的逻辑块号和物理块号。
- i_prealloc_block 和 i_prealloc_count成员，用于数据块的预分配。
- xattr_sem成员，一个读/写semaphore。
- i_acl和i_default_acl成员，指向文件的ACL。

当要处理一个Ext2文件时，超级块的alloc_inode方法是用ext2_alloc_inode()函数来实现的。它首先从ext2_inode_cachep slab分配其中取得一个

ext2_inode_info描述符，接着返回嵌套在这个新ext2_inode_info描述符中的inode对象地址。

18.5. Ext2 Methods

18.5.1. Ext2 Superblock Operations

许多VFS超级块操作在Ext2中有一个特定的实现。这些超级块方法实现的地址保存在ext2_sops结构中。

18.5.2. Ext2 inode Operations

对于常规文件和目录文件，Ext2方法的地址分别保存在ext2_file_inode_operations 和ext2_dir_inode_operations中。

18.5.3. Ext2 File Operations

文件操作的方法保存在ext2_file_operations表中。

18.6. Managing Ext2 Disk Space

在本节中，我们将解释Ext2文件系统是如何管理磁盘空间，如何分配和回收inode和数据块。2个主要的问题必须定位：

- 磁盘空间管理必须尽力避免文件在物理上的不连续。
- 磁盘空间管理必须在时间上高效的；这意味着，内核应该能快速的从文件偏移中取得对应的在Ext2分区中的逻辑磁盘块号。为了做到这一点，内核应该尽可能的限制访问储存在磁盘上的地址表的次数。

18.6.1. Creating inodes

ext2_new_inode()函数创建一个Ext2磁盘inode，返回对应的inode对象地址。函数仔细的选择块组来包含这个新的inode；尽可能的会将文件放在与它的父目录相同的块组中。为了平衡块组中常规文件和目录文件的数量，Ext2引入了一个"debt"参数给每个块组。

函数有两个参数：inode对象所属的目录的inode地址dir。inode创建所用的类型type。第二个参数也包含了MS_SYNCHRONOUS标志位，它将使当前进程挂起，直到inode被分配完成。函数执行下列动作：

1. 调用new_inode()来分配一个新的VFS inode对象；初始化它的i_sb成员为超级块地址dir->i_sb，然后把它添加到in-use inode链表和超级块的链表。
2. 如果新的inode是一个目录，函数调用find_group_orlov()来寻找一个合适的块组。这个函数实现了下列启发式算法：

a. 目录以文件系统的root目录为父目录。那么，函数搜寻所有块组，找到一个含有空闲inode和空闲块数量在平均水平以上的块组。如果没有这样的块组，跳转到2c。

b. 不以文件系统的root目录为父目录的目录，如果满足下列规则就应该放在父目录所在的块组中：
   - 块组没有包含过多目录。
   - 块组有足够数量的空闲inode。
   - 块组有较小的"debt"(一个块组的debt保存在ext2_sb_info-> s_debts

中；当一个新目录加入时增加debt并在另一种类型的文件加入时减少）
如果父目录所在的块组不满足这些规则，它将选择第一个满足它们的块组。如果没有这样的块组存在，跳转到2c。

c. 现在是"fallback"规则，如果没有合适的块组被找到时会调用。函数从父目录所在的块组开始并选择第一个含有超过平均空闲inode数/块组的块组。

3. 如果新的inode不是一个目录，它将调用find_group_other()函数在一个含有空闲inode的块组中分配之。这个函数选择块组，是从其父目录所在块组开始，再逐渐远离；精确的说：

a. 从其父目录dir所在块组开始，执行一个快速对数搜索。算法搜索log(n)个块组，n是块组的总数。算法向前跳转直到找到一个可用的块组。例如，如果我们从块组i开始，算法将考虑块组i mode(n)，i+1 mode(n)，i+1+2 mode(n)，i+1+2+4 mode(n)等等。

b. 如果对数算法搜索未能成功找到包含空闲inode的块组，函数将从包含父目录dir的块组开始，执行一个线性搜索。

4. 调用read_inode_bitmap()得到选定的块组的inode位图并找到第一个空bit，接着得到空闲inode的号码。

5. 分配磁盘inode：设置inode位图中对应bit，并标记包含该位图的buffer为dirty。而且，如果文件系统如果指定了MS_SYNCHRONOUS标志位，函数调用sync_dirty_buffer()开始一个I/O写操作并等待操作的结束。

6. 递减组描述符的bg_free_inodes_count成员。如果新inode是一个目录，函数增加bg_used_dirs_count成员，并标记包含组描述符的buffer为dirty。

7. 增加或减少保存在超级块的s_debts数组的组的计数器，这是根据inode是否是常规文件还是一个目录来决定。

8. 递减ext2_sb_info->s_freeinodes_counter；而且，如果新inode是一个目录，将增加ext2_sb_info->s_dirs_counter。

9. 设置超级块的s_dirt标志位为1，并标记包含它的buffer为dirty。

10. 设置VFS的超级块对象的s_dirt为1。

11. 初始化inode对象的成员。

12. 初始化inode的ACL。

13. 把新inode对象插入到哈希表inode_hashtable中，并调用mark_inode_dirty()把inode对象移动到超级块的dirty inode链表。

14. 调用ext2_preread_inode()从磁盘中读取包含inode的块，并把块放在page cache中。

15. 返回新inode对象的地址。

18.6.2. Deleting inodes

ext2_free_inode()函数删除一个磁盘inode。内核在调用该函数之前，要进行一些清理工作：inode对象要从inode哈希表中移除，链接到该inode的硬链接要从合适的目录删除，文件要被截取为0来回收数据块。函数执行下列动作：

1. 调用clear_inode()，它将顺序执行下列操作：

a. 移除任何与该inode相联系的dirty的"间接"buffer；它们被收集在inode->i_data->private_list中。

b. 如果inode的I_LOCK标志位被设置，说明inode的buffer中的某些正在I/O数据传输中；函数挂起当前进程直到I/O数据传输完成。

c. 如果有定义，调用超级块对象的clear_inode方法；Ext2文件系统没有定义它。

d. 如果inode是与设备文件相关联，它将把inode对象从设备的inode链表中移除。

e. 设置inode的状态为I_CLEAR（这个inode对象不再有意义）。
2. 通过inode号和每个块组包含的inode数量，计算出块组的组号。
3. 调用read_inode_bitmap()得到inode位图。
4. 增加组描述符的bg_free_inodes_count成员。如果删除的是一个目录，它还要递减bg_used_dirs_count成员。标记包含组描述符的buffer为dirty。
5. 如果删除的inode是一个目录，递减ext2_sb_info->s_dirs_counter，设置超级块的s_dirty标志位为1，标记包含它的buffer为dirty。
6. 清楚磁盘inode位图中对应该inode的bit并标记包含该位图的buffer为dirty。而且，如果文件系统在加载时设置了MS_SYNCHRONIZE标志位，它将调用sync_dirty_buffer()等待位图的buffer的写操作的完成。

18.6.3. Data Blocks Addressing

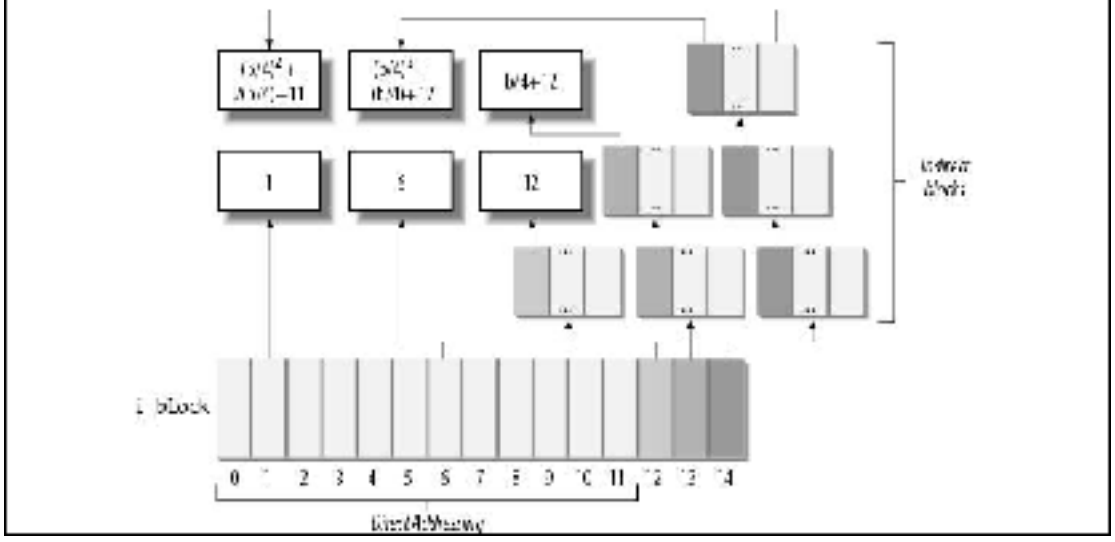
每个非空常规文件由一组数据块组成。这些数据块可以被其在文件中的偏移或在磁盘分区中的逻辑块号来定位。

从文件中的偏移f中取得逻辑块号是2步过程：
1. 从偏移量f中取得文件块号。
2. 把文件块号转换为逻辑块号。

然而，Ext2文件的逻辑数据块在物理上不一定邻接。

在磁盘inode的i_block成员是一个EXT2_N_BLOCKS大小的数组，它包含了逻辑块号到物理块号的转化。这个数组分为4个不同的类型：

Figure 18-5. Data structures used to address the file's data blocks



18.6.5. Allocating a Data Block

当内核需要定位一个包含了所需数据的磁盘块时，它将调用ext2_get_block()函数。如果块不存在，函数自动分配块给文件。记住，这个函数在内核对Ext2常规文件读或写时都可能会调用；函数只会在所需的块不在page cache中时才会被调用。

ext2_get_block()函数在必要时会调用ext2_alloc_block()函数在Ext2分区中寻找一个空闲块。如果有必要，函数也会分配用于间接寻址的块。

为了减少文件碎片，Ext2文件系统尝试在最近分配的块附近分配一个新的块。如果失败

了，文件系统在包含inode的块组中寻找一个新块。最坏情况下，会在其它块组中寻找空闲块。

Ext2文件系统使用了数据块的预分配。文件不仅仅得到请求的块，而会得到一组最多8个邻接的块。ext2_inode_info->i_prealloc_count保存了预分配给文件的还未使用的数据块，i_prealloc_block成员储存了将要被使用的预分配的块的逻辑块号。所有预分配的仍未被使用的块将在文件关闭时，被截断时或写操作不是顺序进行时将被释放。

ext2_alloc_block()函数接收参数：一个指向inode对象的指针，一个goal和一个用来保存错误码的变量的地址。goal是一个逻辑块号，代表了新块期望的位置。ext2_get_block()函数设置goal参数是根据下列启发式规则：

1. 如果将要被分配的块和之前已经分配的块有连续的块号，goal就是前一个逻辑块号加1；从程序的角度看连续的块应该在物理上也是邻接的。
2. 如果第一条规则没有应用，并且之前至少有一个块已经分配给了文件，goal就是这些块的逻辑块号中的一个。更精确的说，它是已经分配出去的逻辑块号。
3. 如果之前的规则都没有应用上，goal就是包含文件的inode的块组的第一个块的逻辑块号。

ext2_alloc_block()函数检查goal是否引用的是之前预分配给文件的块。如果是，它分配对应的块并返回它的逻辑块号；否则，函数丢弃所有预分配的块并调用ext2_new_block()。

ext2_new_block()函数在Ext2分区中按照下列策略搜索空闲块：

1. 如果传递给ext2_alloc_block()函数的goal块是空闲的，函数就分配它。
2. 如果goal块不是空闲的，函数检测是否下一个块是空闲的。
3. 如果goal附近没有空闲块，函数将考虑所有块组，从包含goal的块组开始。对每个块组，函数做下列工作：
   - 查找至少有8个邻接空闲块的组。
   - 如果找不到这样的组，查找有单个空闲块的组。

当一个空闲块被找到，搜索结束。在中止前，ext2_new_block()函数尝试预分配最多8个空闲块，并设置磁盘inode的i_prealloc_block和i_prealloc_count成员。

18.6.6. Releasing a Data Block

当一个进程删除一个文件或截断文件为0长度，所有的数据块必须被回收。这是通过ext2_truncate()完成的，它接收文件的inode对象地址作为参数。函数本质上会扫描磁盘inode的i_block数组来定位所有的数据块。随后将重复调用ext2_free_blocks()释放这些块。

ext2_free_blocks()函数释放一个或多个邻接的数据块的小组。除了在ext2_truncate()中使用，函数主要在释放文件预分配的块时调用。它的参数是：
inode:描述文件的inode对象地址。
block:要被释放的第一个块的逻辑地址。
count:要被释放的邻接的块数量。

函数执行下列动作来释放数据块：

1. 取到包含要被释放的块的块组的块位图。
2. 清除块位图中对应于要被释放的块的bit，标记包含位图的buffer为dirty。

3. 增加块组表述符bg_free_blocks_count成员，标记对应buffer为dirty。
4. 增加磁盘超级块的s_free_blocks_count成员，标记对应buffer为dirtu，设置超级块对象的s_dirt标志。
5. 如果文件系统在加载时设置了MS_SYNCHRONOUS标志位，它将调用sync_dirty_buffer()并等待写位图的buffer操作的结束。

Chapter 19. Process Communication

19.1. Pipes

19.1.1. Using a Pipe

在简单的例子中，pipe只被两个进程所使用。然而，实际上一个pipe可以被任意多个进程使用，那么这就需要使用同步技术（文件所或IPC semaphores）。

19.1.2. Pipe Data Structures

一旦一个pipe被创建，内核将创建一个inode对象和2个文件对象（一个用于读，一个用于写）。

当一个inode对象指向一个pipe，它的i_pipe成员指向一个pipe_inode_info数据结构。

每个pipe还有一个pipe buffer集合。本质上，一个pipe buffer是一个page frame，它要读写的数据。直到Linux 2.6.10，每个pipe仅含有1个pipe buffer。在2.6.11中，pipe(包括FIFO)的数据buffer增加到16个。这个改变大大增强了用户态应用通过pipe读写大块数据的能力。

pipe_inode_info->bufs是一个包含16个pipe_buffer对象的数组，每个对象是:

Type	Field	Description
struct page *	page	Address of the descriptor of the page frame for the pipe buffer
unsigned int	offset	Current position of the significant data inside the page frame
unsigned int	len	Length of the significant data in the pipe buffer
struct pipe_buf_operations *	ops	Address of a table of methods relative to the pipe buffer (NULL if the pipe buffer is empty)

pipe_buffer->ops成员指向anon_pipe_buf_ops表，这个表包含了3个方法:

1. map:在访问pipe buffer中的数据前调用。它简单的对pipe buffer的page frame调用kmap()，以应对pipe buffer被分配了高内存时的情况。
2. unmap:当不再访问pipe buffer中的数据时调用。它调用kunmap()在pipe buffer的page frame。
3. release:当一个pipe buffer将要被释放时调用。这个方法实现了一个单一页面的内存cache:被释放的page frame不是保存了buffer的页面帧，而是一个被

cached page frame，它由pipe_inode_info->tmp_page指向。包含了buffer的page frame变成新的被cached page frame。

这16个pipe buffer能被看成是一个全局循环buffer，为了提高效率，要读取的数据可以跨越多个buffer，然而，写数据时，如果某个buffer中剩下的空间已经不足已完全包含数据，则需要新使用一个buffer，之前的buffer不写入数据。

19.1.2.1. The pipefs special filesystem

一个pipe是作为一个VFS对象实现的，他没有对应的磁盘镜像。在Linux2.6中，它被组织进了pipefs特殊文件系统。这个文件系统没有安装点，因此对用户是不可见的。

init_pipe_fs()函数，在系统初始化时，注册了pipefs文件系统并加载它。

19.1.3. Creating and Destroying a Pipe

pipe()系统调用在内核中是被sys_pipe()函数服务，它调用了do_pipe()函数，这个函数执行了下列步骤：

1. 调用get_pipe_inode()函数，它在pipefs文件系统中分配并初始化了一个inode对象。这个函数执行了下列动作：
   - Pipefs文件系统中分配一个新的inode。
   - 分配一个pipe_inode_info数据结构，并把它的地址保存在inode->i_pipe中。
   - 设置pipe_inode_info->curbuf和pipe_inode_info->nrbufs为0;把bufs数组中所有pipe buffer对象的成员置为0。
   - 初始化pipe_inode_info的r_counter和w_counter成员为1。
   - 设置pipe_inode_info的readers和writers成员为1。
2. 分配一个文件对象和一个文件描述符用于读，设置file对象的f_flag成员为O_RDONLY，初始化f_op成员为read_pipe_fops表地址。
3. 分配一个文件对象和一个文件描述符用于写，设置file对象的f_flag成员为O_WRONLY，初始化f_op成员为write_pipe_fops表地址。
4. 分配一个dentry对象并把它链接到2个file对象和inode对象中。
5. 返回2个file描述符给用户进程。

19.1.4. Reading from a Pipe

对于pipe，读方法的入口在read_pipe_fops表中的pipe_read()函数。

系统调用可能会在下列2种情况中阻塞当前进程：

- 当系统调用开始时，pipe buffer是空的。
- pipe buffer没有包含所有请求的字节数，一个写进程正在睡眠中。

Table 19-3. Reading n bytes from a pipe

	At least one writing process			No writing process
	Blocking read		Nonblocking read	
Pipe Size p	Sleeping writer	No sleeping writer		
p = 0	Copy n bytes and return n, waiting for data when the pipe buffer is empty.	Wait for some data, copy it, and return its size.	Return -EAGAIN.	Return 0.
0 < p < n	Copy p bytes and return p: 0 bytes are left in the pipe buffer.			
p	Copy n bytes and return n: p-n bytes are left in the pipe buffer.			
n				

函数执行了下列操作:

1. 请求inode->i_sem信号量。
2. 读取pipe_inode_info->nrbufs，判断其是否为0;如果它等于0，所有的buffer都是空的。在这种情况下，它需要决定函数是返回或者进程必须阻塞直到另一个进程写进pipe一些数据。I/O操作的类型(阻塞或非阻塞)是被O_NONBLOCK flag标记。如果当前进程必须被阻塞，函数执行下列动作:
   - 调用prepare_to_wait()把当前进程加入到pipe的等待队列中。
   - 释放inode semaphore。
   - 调用schedule()。
   - 一旦被唤醒，调用finish_wait()把当前进程从等待队列中移除，再次请求i_sem，跳转到步骤2。
3. 读取pipe_inode_info->curbuf。
4. 执行pipe buffer的map方法。
5. 拷贝请求的字节数或者pipe buffer中可用的字节数到用户空间。
6. 执行pipe buffer的unmap方法。
7. 更新pipe_buffer对象的offset和len成员。
8. 如果pipe buffer已经变空了(pipe_buffer->len为0)，调用pipe buffer的realse方法来释放对应的page frame，设置pipe_buffer->ops为NULL，移动pipe_inode_info->curbuf，递减nrbufs。
9. 如果所有被请求的字节都被拷贝了，跳转到步骤12。
10. 这里，不是所有的请求字节都被拷贝到用户地址空间。如果pipe的大小大于0，跳转到步骤3。
11. 这里，pipe buffer中没有字节存留。如果至少有一个写进程在睡眠中(pipe_inode_info的waiting_writers大于0)，并且读操作是被阻塞了，它将调用wake_up_interruptbile_sync()来唤醒所有的pipe中等待队列的睡眠进

程，跳回步骤2。
12. 释放i_sem。
13. 调用wake_up_interruptbile_sync()唤醒所有的在等待队列中的写进程。
14. 返回拷贝到用户空间的字节数。

19.1.5. Writing into a Pipe

进程最终是通过调用write_pipe_fops表中的pipe_write()函数来把数据写入pipe。

特别的，如果是对比较少字节的写操作就必须原子的完成。精确的说，如果两个或更多个进程并发的写进一个pipe。每次写操作的数据少于4096字节（pipe buffer大小）必须原子的完成，而不会有交错。然而，写大于4096字节可能会是非原子的，可能会使进程进入睡眠。

而且，如果pipe没有读进程了(如果readers等于0)每个写操作将会失败。在这种情况下，内核发送SIGPIPE信号给写进程并终止write()系统调用，返回-EPIPE错误码。这就是大家熟悉的"Broken pipe"信息。

pipe_write()函数执行下列操作:
1. 获取i_sem信号量。
2. 检测pipe是否最少有一个读进程。如果没有，它发送一个SIGPIPE信号给current进程，释放inode semaphore，并返回一个-EPIPE值。
3. 计算出最后一个写pipe buffer的索引。如果pipe buffer有足够多的剩余空间来保存要被写入的字节，那么拷贝数据进去:
   - 执行pipe buffer的map方法。
   - 把所有的数据拷贝进pipe buffer中。
   - 执行unmap方法。
   - 更新对应pipe_buffer对象的len成员。
   - 跳转到步骤11。
4. 如果pipe_inode_info->nrbufs等于16，那么说明没有空的pipe buffer可以来保存要被写入的数据。在这种情况下:
   - 如果写操作是非阻塞的，跳转到步骤11并返回-EAGAIN错误码。
   - 如果写操作是阻塞的，它对pipe_info->waiting_writers加一，调用prepare_to_wait()来把current添加到pipe的等待队列中，释放inode semaphore，调用schedule()。一旦唤醒了，它调用finish_wait()把current进程从等待队列中移除，再次请求inode semaphore，递减waiting_writers成员，跳回步骤4。
5. 现在，至少有一个pipe buffer。计算出第一个空的pipe buffer的索引。
6. 分配一个新的page frame，除非pipe_inode_info->tmp_page为非NULL。
7. 从用户空间拷贝最多4096字节到page frame中。
8. 更新pipe_buffer对象的某些成员，设置page成员为刚分配的page frame描述符地址，ops成员为anon_pipe_buf_ops表的地址，offset成员为0，len成员为写入的字节数目。
9. 增加nrbufs计数器。
10. 如果不是所有请求的字节都被写入，跳转回步骤4。
11. 释放inode semaphore。
12. 唤醒所有睡眠在等待队列中的读进程。

13. 返回写入pipe buffer中的字节数。

19.2. FIFOs

在Linux2.6中，FIFO和pipe几乎是相同的，它们都使用了pipe_inode_info结构。事实上，一个FIFO的read和write操作都是通过pipe_read()和pipe_write()函数来实现的。实际上，它们只有2个重要的不同点：

- FIFO inode出现在系统目录中，而不是出现在pipefs特殊文件系统中。
- FIFO是一个双向通信通道。

19.2.1. Creating and Opening a FIFO

一个进程通过mknod()系统调用来创建一个FIFO。

当一个进程打开一个FIFO，VFS会执行类似于对设备文件那样的操作。与FIFO相联系的inode对象的初始化是通过一个文件系统相关的read_inode超级块方法完成；这个方法检查了磁盘上的inode是否代表了一个特殊文件，如果有必要则调用init_special_inode()函数，这个函数会设置inode对象的i_op成员为def_fifo_inode表的地址。随后，内核设置file对象的文件操作表为def_fifo_inode，并执行表中的open方法，它实际上是由fifo_open()实现的。

fifo_open()函数初始化了特定于FIFO的数据结构；它执行下列操作：

1. 请求i_sem inode semaphore。
2. 检测inode对象的i_pipe成员；如果它是NULL，分配并初始化一个新的pipe_inode_nifo结构。
3. 根据open()系统调用的参数中指定的访问权限，初始化file对象的f_op成员为合适的文件操作表地址。
4. 如果访问模式是只读或可读写，函数将把pipe_inode_info结构的readers和r_counter成员加1。而且，如果访问模式是只读并且没有其他读进程，它将唤醒任何的正在睡眠中的写进程。

Table 19-5. FIFO's file operations

Access type	File operations	Read method	Write method
Read-only	read_fifo_fops	pipe_read( )	bad_pipe_w( )
Write-only	write_fifo_fops	bad_pipe_r( )	pipe_write( )
Read/write	rdwr_fifo_fops	pipe_read( )	pipe_write( )

5. 如果访问模式是只写或可读写，函数将把pipe_inode_info结构的writers和w_counter成员加1。而且，如果访问模式是只读并且没有其他读进程，它将唤醒任何的正在睡眠中的写进程。
6. 如果没有读者或写着，函数决定是否阻塞或返回一个错误码而结束。

Table 19-6. Behavior of the fifo_open( ) function

Access type	Blocking	Nonblocking
Read-only, with writers	Successfully return	Successfully return

Table 19-6. Behavior of the fifo_open( ) function

Access type	Blocking	Nonblocking
Read-only, no writer	Wait for a writer	Successfully return
Write-only, with readers	Successfully return	Successfully return
Write-only, no reader	Wait for a reader	Return -ENXIO
Read/write	Successfully return	Successfully return

7. 释放inode semaphore并结束，返回0（成功）。

19.3. System V IPC

当一个进程请求一个IPC资源时，IPC数据结构被动态创建。一个IPC资源是持久的：除非被进程显示的移除，它将存在于内存中直到系统关闭。一个IPC资源可以被每个进程使用。

每个新的IPC资源被一个32-bit IPC KEY标记，它类似于文件路径。每个IPC资源也有一个32-bit IPC identifier，它类似于与一个打开相联系的文件描述符。IPC identifier被内核分配给IPC资源并在系统中唯一，IPC key能被程序员自由的选择。

当有2个或更多进程希望通过IPC资源通信，它们需要都引用该资源的IPC identifier。

19.3.1. Using an IPC Resource

IPC资源通过调用semget()，msgget()或shmget()函数来获取。

这3个函数主要目标是从IPC key（函数的第一个参数）中，计算出对应的IPC identifier。如果没有IPC资源已经与IPC key相联系，一个新资源被创建。

假设2个独立的进程希望共享一个IPC进程。这能通过2个可能的方法完成：

- 2个进程使用一个预先定义好的IPC key。这是最简单的情况，然而，有可能该IPC key被另一个不相关的程序选用了。在这种情况下，IPC函数可能会成功调用并返回一个错误资源的IPC identifier。
- 一个进程调用semget()，msgget()或shmget()函数时，用IPC_PRIVATE作为它的IPC key。一个新的IPC资源被创建，进程能通过IPC identifier或fork另一个进程来通信。这确保了IPC资源不会被其它应用偶然使用。

每种类型的IPC资源都拥有一个ipc_ids数据结构。

Table 19-8. The fields of the ipc_ids data structure

Type	Field	Description
int	in_use	Number of allocated IPC resources
int	max_id	Maximum slot index in use
unsigned short	seq	Slot usage sequence number for the next allocation

Table 19-8. The fields of the ipc_ids data structure

Type	Field	Description
unsigned short	seq_max	Maximum slot usage sequence number
struct semaphore	sem	Semaphore protecting the ipc_ids data structure
struct ipc_id_ary	nullentry	Fake data structure pointed to by the entries field if this IPC resource cannot be initialized (normally not used)
struct ipc_id_ary *	entries	Pointer to the ipc_id_ary data structure for this resource

ipc_id_ary数据结构又两个成员:p和size。P成员是一个指针数组，其中每个指针指向kern_ipc_perm数据结构，每个代表一个可分配的资源。Size成员是该数组的大小。

每个kern_ipc_perm数据结构与一个IPC资源相联系。

Table 19-9. The fields in the kern_ipc_ perm structure

Type	Field	Description
spinlock_t	lock	Spin lock protecting the IPC resource descriptor
int	deleted	Flag set if the resource has been released
int	key	IPC key
unsigned int	uid	Owner user ID
unsigned int	gid	Owner group ID
unsigned int	cuid	Creator user ID
unsigned int	cgid	Creator group ID
unsigned short	mode	Permission bit mask
unsigned long	seq	Slot usage sequence number
void *	security	Pointer to a security structure (used by SELinux)

19.3.3. IPC Semaphores

实际上，IPC semaphore比内核semaphore要更难处理写，因为一下2个主要原因:

- 每个IPC semaphore是一个semaphore值的集合，并不像内核semaphore是一个单一的值。这意味着同一个IPC资源能保护几个独立的被共享的数据结构。每个IPC semaphore中的semaphore数量必须作为semget()的参数来被指定。
- System V IPC semaphore提供了一种fail-safe机制来应对：一个进程死去时，却没有undo之前发出的对semaphore的操作。当一个进程选择使用这个机制时，最终的操作被称为undoable semaphore操作。

sem_ids变量保存了ipc_ids数据结构；对应的ipc_id_ary数据结构包含了一个指针数组，每个指针指向了sem_array数据结构，每项代表了一个IPC semaphore资源。

sem_ids.entries数组保存的指针是指向kern_ipc_perm数据结构，每个这种数据结构是sem_array数据结构的第一个成员。sem_array数据结构为：

Table 19-10. The fields in the sem_array data structure

Type	Field	Description
struct kern_ipc_perm	sem_perm	kern_ipc_perm data structure
long	sem_otime	Timestamp of last semop( )
long	sem_ctime	Timestamp of last change
struct sem *	sem_base	Pointer to first sem structure
struct sem_queue *	sem_pending	Pending operations
struct sem_queue **	sem_pending_last	Last pending operation
struct sem_undo *	undo	Undo requests
unsigned long	sem_nsems	Number of semaphores in array

19.3.4. IPC Messages

一个进程可以产生一个消息到一个IPC message queue，它将等待另一个进程来读取它。

一个消息由一个固定大小的头部和一个可变长度的文本组成；它能被一个整数所标记，这就允许一个进程有选择的从它的消息队列中提取信息。一旦一个进程已经从一个IPC message queue中读取了一个信息，内核将销毁该信息。

全局变量msg_ids保存了ipc_ids数据结构；对应的ipc_id_ary数据结构包含了一个指向shmid_kernel数据结构的指针数组。数组保存的指针是指向kern_ipc_perm数据结构，每个结构是msg_queue数据结构的第一个成员。

Table 19-12. The msg_queue data structure

Type	Field	Description
struct kern_ipc_perm	q_perm	kern_ipc_perm data structure
long	q_stime	Time of last msgsnd( )
long	q_rtime	Time of last msgrcv( )
long	q_ctime	Last change time
unsigned long	q_qcbytes	Number of bytes in queue

Table 19-12. The msg_queue data structure

Type	Field	Description
unsigned long	q_qnum	Number of messages in queue
unsigned long	q_qbytes	Maximum number of bytes in queue
int	q_lspid	PID of last msgsnd( )
int	q_lrpid	PID of last msgrcv( )
struct list_head	q_messages	List of messages in queue
struct list_head	q_receivers	List of processes receiving messages
struct list_head	q_senders	List of processes sending messages

其中最重要的成员是q_messages，它是队列中message链表的头部。

每个message被分割成一个或多个页面，它们是动态分配的。第一个页面的开始处保存的是message头部，它的数据结构类型是msg_msg：

Type	Field	Description
struct list_head	m_list	Pointers for message list
long	m_type	Message type
int	m_ts	Message text size
struct msg_msgseg *	next	Next portion of the message
void *	security	Pointer to a security data structure (used by SELinux)

m_list成员保存的是指向前一个和下一个message的指针。message正文紧跟在msg_msg描述符的后面；如果message的长度大于4072字节(页面大小减去msg_msg描述符的大小)，就会再分配一个新的页面，新页面的地址保存在msg_msg->next中。第二个页面的开始处是一个msg_msgseg类型的描述符，它的next指针可选的指向了第三个页面的地址。

19.3.5. IPC Shared Memory

每个希望访问保存在IPC shared memory region中的数据结构的进程，必须在它的地址空间中增加一个新的memory region，这个memory region将映射与IPC shared memory region相关的page frame。

shmget()函数被调用来得到一个共享memory region的IPC标识符，或者创建它。

shm_ids变量保存的是IPC shared memory resource类型的ipc_ids数据结构；对应的ipc_id_ary数据结构包含了指向shmid_kernel数据结构指针数组，每一项指向了一个IPC shared memory resource。同样的，指针指向的是kern_ipc_perm数据结构。

Table 19-14. The fields in the shmid_kernel data structure

Type	Field	Description
struct kern_ipc_perm	shm_perm	kern_ipc_perm data structure
struct file *	shm_file	Special file of the segment
int	id	Slot index of the segment
unsigned long	shm_nattch	Number of current attaches
unsigned long	shm_segsz	Segment size in bytes
long	shm_atim	Last access time
long	shm_dtim	Last detach time
long	shm_ctim	Last change time
int	shm_cprid	PID of creator
int	shm_lprid	PID of last accessing process
struct user_struct *	mlock_user	Pointer to the user_struct descriptor of the user that locked in RAM the shared memory resource (see the section "The clone( ), fork( ), and vfork( ) System Calls" in Chapter 3)

最重要的成员是shm_file，它保存了一个file对象地址。这一点反映了IPC与VFS层的紧密联系。每个IPC shared memory region与一个属于shm特殊文件系统的file相联系。

因为shm文件系统在系统目录中没有安装点，所以没有用户可以通过VFS系统调用来打开和访问其中的文件。而且，当一个进程"attaches"一个段，内核调用do_mmap()并创建一个新的对该文件的共享内存映射。所以，属于shm特殊文件系统的file仅只有一个方法：mmap，它是用shm_mmap()函数实现的。

在之前描述过，一个对应于IPC的memory region是被一个vm_area_struct对象描述的；它的vm_file成员指针指回特殊文件系统中的file对象，接着因用了一个dentry对象和一个inode对象。保存在inode->i_ino成员中的inode号，实际上是IPC shared memory region中的slot index。因此，inode对象间接的引用了shmid_kernel描述符。

19.3.5.2. Demand paging for IPC shared memory regions

用于IPC shared memory 的memory regions总是定义了nopage方法。它是被shmem_nopage()函数实现的，它执行了下列操作：

1. 遍历VFS对象指针链表，获取IPC shared memory resource的inode对象的地址。
2. 由memory region的vm_start成员计算出逻辑页面数量和请求的地址。
3. 检查页面是否已经包含在了page cache中；如果是，结束并返回它的描述符地址。
4. 检测页面是否已经包含在了swap cache中并已经是最新的了；如果是，结束并返回它的描述符地址。

5. 检测内嵌在inode对象中的shmem_inode_info是否储存了一个被swapped-out的页面标识符。如果是，它调用read_swap_cache_async()来把页面swap-in，函数将等待直到数据传输完成，然后结束并返回页面描述符地址。
6. 否则，页面没有保存在swap area中；所以，函数从buddy系统中分配一个新的页面，插入到page cache并返回其地址。

Chapter 20. Program ExZecution

20.1. Executable Files

当一个进程开始执行一个新的程序，它的执行环境发生了剧烈的变动，因为之前的资源将被丢弃，进程的特权级也有可能发生变化。然而，新进程的PID并没有改变，打开的文件描述符也会继承下来。

20.1.1. Process Credentials and Capabilities

传统上说，Unix系统中每个进程有一些credentials，它把进程绑定到一个特定用户和一个特定的用户组。Credentials在多用户系统中是十分重要的，因为它们决定了进程能做什么，不能做什么。

Credentials的使用需要在进程数据结构中和被保护的资源中有支持。一个常用的资源是文件。在Ext2文件系统，一个文件被一个特定用户拥有并且被绑定到一个用户组。当一个进程尝试访问一个文件，VFS总是会检测访问是否是合法的。

Table 20-1. Traditional process credentials

Name	Description
uid, gid	User and group real identifiers
euid, egid	User and group effective identifiers
fsuid, fsgid	User and group effective identifiers for file access
groups	Supplementary group identifiers
suid, sgid	User and group saved identifiers

UID为0表明是超级用户(root)，GID为0代表root组。如果一个进程的credentials中的值为0，内核将绕过权限检测。

当一个进程被创建时，它总是继承它的父进程的credentials。然而，这些credentials在随后能被修改，要么当在进程执行新程序时或当它发出合适的系统调用时。通常，进程的uid,euid,fsuid和suid包含了同样的值。当进程执行一个setuid程

序，euid和fsuid被设置为文件拥有者的值。几乎所有的检测使用这两个成员：fsuid被用来文件相关的操作，euid被用在其它所有操作。

为了说明fsuid是如何使用的，考虑当一个用户想改变它的密码时。所有的密码保存在一个文件中，但它不能直接修改文件，因为它是被保护的。所以，它调用了一个系统程序/usr/bin/passwd，这个程序设置了setuid标志并且它的拥有者是超级用户。当进程执行这个程序时，它的euid和fsuid被设置为0。现在进程能访问该文件了，因为当内核执行访问操作时，它发现fsuid为0。当然，/usr/bin/passwd程序只允许用户修改它自己的密码。

20.1.1.1. Process capabilities

POSIX.1e 描述了另一种进程credentials模型，它是基于概念"capabilities"。Linux内核支持POSIX capabilities,但是大多数Linux发布版并不使用它。

一个capability简单的是一个flag，它指明了进程是否允许执行一个特定操作或一类特定操作。

VFS和Ext2文件系统目前都不支持capability模型，因此没有办法可以把一个可执行文件和capability联系在一起。然而，进程能显示的得到或降低它的capability，通过capget()和capset()系统调用。

20.1.1.2. The Linux Security Modules framework

在Linux2.6中，capability被紧密的整合进了Linux Security Modules(LSM)中。简而言之，LSM框架允许开发者为内核安全定义许多可选的模型。

每个安全模型被一个security hooks集合来实现。一个security hook是一个函数，当内核将执行一个重要的，安全相关的操作时会调用。这个hook函数决定操作是应该被执行还是被拒绝。

security hooks被保存在security_operations类型的表中。Hook表的地址保存在security_ops变量中。缺省的，内核通过dummy_security_ops表来实现最小限度的安全模型；表中的每个hook本质上将检测对应的capability或者无条件的返回0(操作允许)。

例如，stime()和settimeofday()函数在改变系统日期和时间前，会调用settime security hook。dummy_security_ops表中对应函数将检测当前进程是否有CAP_SYS_TIME能力，返回0或-EPERM。

20.1.2. Command-Line Arguments and Shell Environment

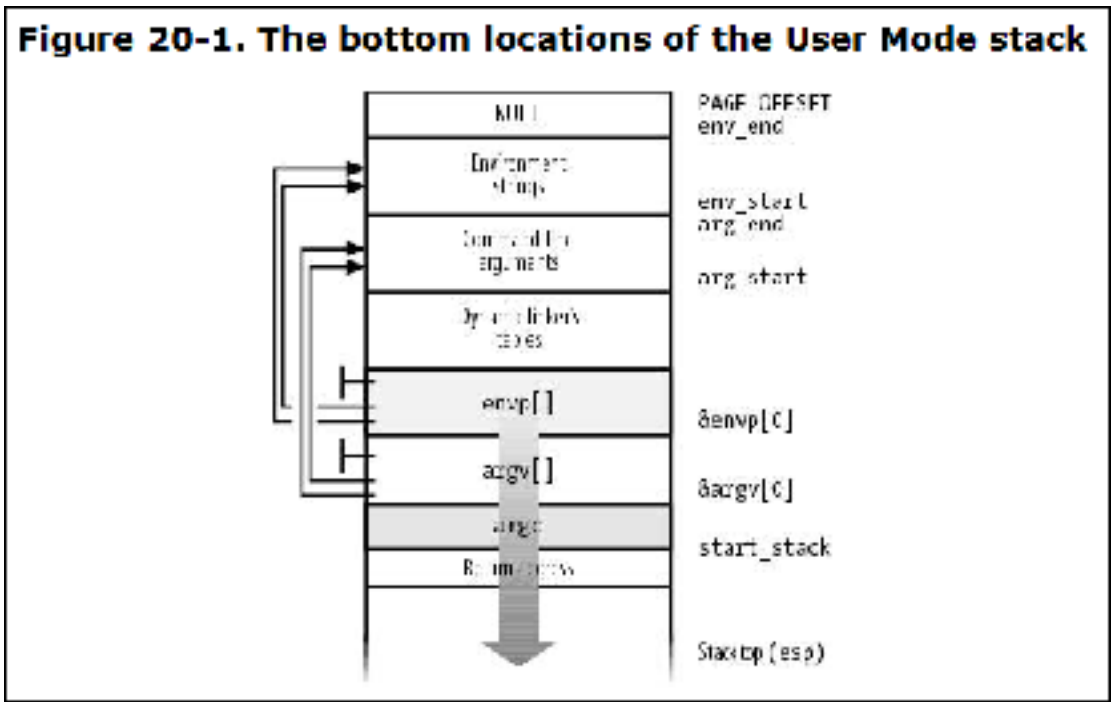


Figure 20-1. The bottom locations of the User Mode stack

20.4. The exec Functions

sys_execve()服务程序接收下列参数(都在用户控件):

- 可执行文件路径名的地址。
- 命令行参数数组地址。
- 环境变量数组地址。

函数拷贝可执行文件路径名到一个新分配的page frame中。它接着调用do_execve()函数，传入page frame的地址，指针数组，用户态寄存器的内容保存在内核栈中的地址。do_execve()执行下列操作:

1. 动态分配一个linux_binprm数据结构，它之后将被填充与可执行文件相关的数据。
2. 调用path_lookup(),dentry_open()和path_release()得到dentry对象，file对象，inode对象。如果失败了，它返回合适的错误码。
3. 验证当前进程是否可执行；也检查文件没有正在被写入，这通过检测inode的i_writecount成员；如果是-1，表明写操作被禁止。
4. 在多处理器系统中，它调用sched_exec()函数来决定负载最轻的CPU来执行这个新程序，并且把当前进程迁移到那去。
5. 调用init_new_context()来检测当前进程是否使用了专用LDT；在这种情况下，函数分配和填充一个新的LDT。
6. 调用prepare_binprm()函数来填充linux_binprm数据结构。这个函数执行下列操作:
   - 再次检测文件是否是可执行的；如果不是，就返回一个错误码。
   - 初始化linux_binprm结构的e_uid和e_gid成员。
   - 填充linux_binprm结构的buf成员，使用可执行文件的头128字节。这些字节包含了魔数和其它信息。
7. 拷贝文件名，命令行参数，环境变量到一个或多个新分配的page frame。

8. 调用search_binary_handler()函数，参数是linux_binprm数据结构，它扫描formats链表并尝试应用每个元素的load_binary方法。对formats的扫描当一个load_binary方法成功的识别出文件的可执行类型。
9. 如果可执行文件格式不再format链表中，它将释放所有分配的page frames并返回错误码-ENOEXEC。Linux不能识别可执行文件类型。
10. 否则，函数释放linux_binprm数据结构并返回从load_binary方法的到的返回码。

对应可执行文件的load_binary方法将执行下列操作(我们假设可执行文件保存在文件系统中，它允许文件内存映射并且它需要一个或多个共享库)：

1. 检测保存在的头128字节中的魔数，它标明了可执行文件格式。如果魔数不匹配，返回错误码-ENOEXEC。
2. 读取可执行文件的头部。它描述了程序的段和需要的共享库。
3. 从可执行文件中取得动态链接器地址，它用来定位共享库并映射到内存中。
4. 得到动态链接器的dentry对象。
5. 检测动态链接器的执行权限。
6. 拷贝动态链接器的头128字节到buffer中。
7. 对动态链接器类型执行一些连续的检测。
8. 调用flush_old_exec()函数来释放之前使用的几乎所有的资源；这个函数执行下列操作：
   - 如果信号处理函数表被其他进程共享，他分配一个新的表并递减旧表的使用计数；而且，它把进程从就得线程组中取出。这些操作是通过调用de_thread()函数。
   - 如果files_struct被其他进程所共享，调用unshare_files()来生成对files_struct结构的一个拷贝。
   - 调用exec_mmap()函数来释放memory描述符，所有的memory regions，和所有分配给进程的page frames以及清空进程的Page Tables。
   - 设置进程描述符的comm成员为可执行文件路径名。
   - 调用flush_thread()函数来清空保存在TSS段中的浮点寄存器和debug寄存器。
   - 更新信号处理函数表，把每个信号重置为它的缺省动作。这是通过调用flush_signal_handlers()函数。
   - 调用flush_old_files()函数来关闭所有对应于files->close_on_exec的打开文件。
9. 清空进程描述符中的PF_FORKNOEXEC标志。这个标志在进程被fork时设置当它执行一个新的程序时被清空。
10. 设置新的进程描述符中的personality成员。
11. 调用arch_pick_mmap_layout()选择进程的memory regions布局。
12. 调用setup_arg_pages()函数来分配一个新的memory regions描述符给用户栈，并把它插入进程的地址空间。setup_arg_pages()也分配page frames来包含命令行参数和环境变量到新的memory regions。
13. 调用do_mmap()函数来创建一个新的memory region来映射可执行文件的代码段。memory region的初始线性地址依赖与可执行文件的格式。ELF程序是从现行地址0x08048000开始装载。
14. 调用do_mmap()函数创建一个新memory region来映射可执行文件的数据段。
15. 分配额外的memory region给可执行文件的其它特殊段。通常是没有。

16. 调用函数装载动态链接器。如果动态链接器是一个ELF类型的，函数名为load_elf_interp()。
17. 把linux_binfmt对象的地址保存在进程描述符的binfmt成员中。
18. 确定进程的新capabilities。
19. 创建特定动态链接器表并把它们保存在用户栈中。
20. 设置进程的memory描述符的start_code, end_code, start_data, end_data, start_brk, brk, 和 start_stack的值。
21. 调用do_brk()函数来创建一个匿名memory region来映射程序的bss段。
22. 调用start_thread()宏修改用户态寄存器eip和esp保存在内核栈中的值，使它们指向动态链接器的入口点和新用户栈的顶部。
23. 如果进程被跟踪了，它通知debugger关于execve()系统调用的完成。
24. 返回0(成功)。

新程序并不能被马上执行，因为动态链接器必须加载共享库。

虽然动态链接库运行在用户态，我们简单的勾画出它是如何执行的。它首先的工作是为自己建立一个基本的执行环境，从用户栈中环境变量和arg_start之间的信息开始。接着动态链接器必须检测程序使用了哪个动态链接库以及库中哪个函数被请求了。接下来，解释器发出mmap()系统调用来创建memory region映射包含了请求的库函数的页面。接着，解释器更新所有对共享库的符号的引用，这是根据库的memory regions的线性地址。最后动态链接器结束它的执行，跳转到程序的main函数入口点来执行。从此以后，进程开始执行可执行文件和共享库的代码。

Appendix A. System Startup

A.1. Prehistoric Age: the BIOS

在电脑打开电源的时刻，它实际上是无法使用的，因为RAM芯片中包含的是随机数据并且操作系统还未运行。为了开始启动过程，一个特殊的硬件电路把CPU的RESET pin置位。在RESET置位后，处理器的一些寄存器(包括cs和eip)被设置成固定的值，位于物理地址0xfffffff0的代码将被执行。这个地址被硬件映射到某个只读，持久的内存芯片中，它通常被称作Read-Only Memory (ROM)。保存在ROM中的程序在80X86体系中通常被称为Basic Input/Output System (BIOS)，因为其包含了许多中断驱动的低级过程，被所有操作系统用在启动阶段处理硬件设备。

一旦进入保护模式，Linux将不再使用BIOS，Linux将为每个硬件设备提供自己的设备驱动。事实上，BIOS过程必须在实模式下执行，因此它们不能共享函数。

BIOS使用实模式，因为在电脑开机时它是唯一可用的模式。一个实模式地址由一个seq段和一个off偏移量组成；对应的物理地址由seg*16+off计算出来。于是，没有使用Global Descriptor Table, Local Descriptor Table, or paging table。值得注意的是，初始化GDT,LDT和paging table的代码必须在实模式下运行。

Linux在启动阶段必须从磁盘或其它外部设备中取得内核镜像时，会强制使用BIOS。BIOS bootstrap过程本质上执行下列4个操作:

1. 对电脑硬件执行一系列测试，确认哪些设备存在，它们是否正常工作。这个过程通常被称作Power-On Self-Test (POST)。在这个阶段，许多信息，例如BIOS版本，将被显示出来。
   近期的80X86，AMD64和Itanium电脑使用了Advanced Configuration and Power Interface(ACPI)标准。在兼容ACPI的BIOS的bootstrap代码会建立许多表，它们会描述系统中的硬件设备。这些表的格式是独立的，能被操作系统内核读取，从而学习如何处理设备。
2. 初始化硬件设备。这个阶段在现代基于PCI体系中是很关键的，因为它要保证所有硬件设备操作不会发生IRQ线和I/O port的冲突。在这个阶段的末尾，一个已安装的PCI设备的表将被显示。
3. 搜索一个用于启动的操作系统。事实上，依赖与BIOS的设置，程序要尝试搜索每个软盘，磁盘和CD-ROM的第一个扇区。
4. 一旦一个合法的设备被找到，它将把第一个扇区的内容拷贝到RAM中，从物理地址0x00007c00开始，并跳转到这个地址并执行刚刚加载的代码。

A.2. Ancient Age: the Boot Loader

Boot loader是一个被BIOS调用的程序，用来把操作系统镜像加载进RAM。让我们简单的勾画boot loader是如何在IBM PC体系中工作的。

为了从软盘中启动，保存在第一个扇区的指令被加载进RAM并被执行；这些指令拷贝所有包含内核镜像的扇区进RAM。

从磁盘启动与从软盘不同。磁盘的第一个扇区被称作Master Boot Record (MBR),包含了分区表和一个小程序，它将加载包含了操作系统的分区的第一个扇区。随后我们将看到，Linux把保存在MBR中的这个简陋的程序用"boot loader"代替，它允许用户选则将要被启动的操作系统。

A.2.1. Booting Linux from a Disk

从磁盘中启动一个Linux内核需要一个两阶段的boot loader。在80X86系统中一个有名的boot loader叫做Linux Loader(LILO)。

LILO可能安装在MBR（替代加载活动分区中的启动扇区的那个小程序）中，或者在每个磁盘分区的启动扇区中。在每种情况下，最终结果是相同的：当loader在启动时执行时，用户可以选择哪个操作系统将被加载。

事实上，LILO boot loader太大了而不能放进一个单一的扇区中，因此它被分成两个部分。MBR或分区启动扇区包含了一个小的boot loader,它被BIOS加载进物理地址0x00007c00开始的RAM中。这个小程序把自己移动到地址0x00096a00，建立实模式栈(范围从0x00098000到0x000969ff)，加载LILO的第二个部分进从地址0x00096c00开始的RAM中，并跳转到那里。

LILO的第二个部分从磁盘中读取一个可启动操作系统的map，提供给用户来选择启动其中之一。最后，当用户选择了要启动的内核，boot loader将要么拷贝对应分区的启动分区进RAM并执行之，要么直接拷贝内核镜像进RAM。

假设一个Linux内核镜像将被被启动了，LILO执行下列操作：

1. 调用BIOS程序来显示一个”Loading”信息。
2. 调用一个BIOS程序来从磁盘中加载内核镜像的初始部分：内核镜像的头512字节被加载进地址0x00090000，同时setup()函数代码被加载进RAM的地址0x00090200。
3. 调用一个BIOS程序加载内核的其余部分进RAM，放在地址0x00010000(对应用make zImage编译的小内核镜像)或0x00100000(对应用make bzImage编译的大内核镜像)。在接下来的讨论中，我们说内核镜像在RAM中是"loaded low" 或 "loaded high"。
4. 跳转到setup()代码。

A.3. Middle Ages: the setup( ) Function

setup()汇编代码被链接在内核镜像文件的0x200偏移处。所以，boot loader能很容易的定位这个代码并拷贝它到RAM的物理地址0x00090200处。

setup()函数必须初始化电脑的硬件设备并建立执行内核程序的环境。虽然BIOS已经初始化了大多数硬件设备，但Linux不依赖它，而是用自己的方法来重新初始化它们，这样可以增加可移植性和健壮性。setup()本质上执行下列操作:

1. 在ACPI兼容的系统中，它调用BIOS程序来在RAM中建立一个表，描述了系统的物理内存布局。在老一些的系统中，它调用一个BIOS程序返回系统可用内存的总量。
2. 设置键盘重复延时和速率。
3. 初始化视频卡。
4. 重新初始化磁盘控制器并确定硬盘参数。
5. 检测IBM Micro Channel bus (MCA)。
6. 检测PS/2鼠标。
7. 检测Advanced Power Management (APM) BIOS支持。
8. 如果BIOS支持Enhanced Disk Drive Services (EDD)，它调用合适的BIOS程序来在RAM中建立一个表，描述系统中的可用磁盘。
9. 如果内核镜像被加载在loaded low处，函数把它移动到0x00001000处。相反地，如果内核镜像在loader high处，函数不做移动。这个步骤是必要的，因为这样就可以把内核镜像保存在一个软盘中并能减少启动时间，保存在磁盘上的内核镜像是压缩了的，解压缩程序需要一些空闲空间来做临时的buffer。
10. 设置8042键盘控制器的A20 pin。A20 pin在切换到保护模式前必须被适当的设置，否则每个物理地址的第21个bit将总是被CPU看成是0。设置A20 pin是一个凌乱的操作。
11. 设置一个临时的Interrupt Descriptor Table(IDT)和一个临时的Global Descriptor Table (GDT).
12. 如果有，则重置floating-point unit (FPU)。
13. 重编程Programmable Interrupt Controllers (PIC)来屏蔽所有中断，除了IRQ2，它用于2个PIC的级联。
14. 把CPU从实模式切换到保护模式，这是通过设置cr0状态寄存器的PE bit。cr0寄存器的PG bit被清零，因此分页仍然被关闭着。
15. 跳转到startup_32()汇编语言函数处。

A.4. Renaissance: the startup_32( ) Functions

有两个不同的startup_32()函数；我们将涉及的是在arch/i386/boot/compressed/head.S文件中。在setup()结束后，函数已经被移动到了0x00100000或0x00001000处，依赖于内核镜像是加载在RAM的high或low处。

函数执行下列操作：
1. 初始化段寄存器和一个临时堆栈。
2. 清除eflags寄存器中的所有bit。
3. 把_edata和_end符号指定的内核数据填充为0。
4. 调用decompress_kernel()函数来解压缩内核镜像。如果内核镜像是loaded low，解压缩后的内核放置在物理地址0x00100000处。否则，如果内核镜像是loaded high，解压缩后的内核被放置在紧跟在压缩镜像后的临时buffer中。解压缩后的镜像接着被移动到最终位置：物理地址0x00100000处。
5. 跳转到物理地址0x00100000处。

解压缩后的内核镜像开始执行另一个startup32()函数，它位于arch/i386/kernel/head.S文件中。使用2个同名函数不会产生任何问题，因为它们是通过跳转到初始物理地址来执行对应函数的。

第二个startup_32()函数为第一个Linux(0号进程)进程建立了执行环境。函数执行下列操作：
1. 用最终值初始化段寄存器。
2. 把内核的bss段填充为0。
3. 初始化保存在swapper_pg_dir和pg0中的临时内核Page Tables，用来把线性地址映射为物理地址。
4. 保存Page Global Directory地址到cr3寄存器中，并通过设置cr0寄存器的PG bit来开启分页功能。
5. 为0号进程建立内核态堆栈。
6. 函数再一次清空eflags寄存器中的所有bit。
7. 调用setup_idt()来填充IDT为空的中断处理函数。
8. 把从BIOS中获得的系统参数和作为参数传递给操作系统的参数放在第一个页面帧中。
9. 识别处理器的模型。
10. 把GDT和IDT表的地址加载进gdtr和idtr寄存器中。
11. 跳转到start_kernel()函数。

A.5. Modern Age: the start_kernel( ) Function

start_kernel()函数将完成Linux内核的初始化工作。几乎所有的内核组件将被该函数初始化；我们只涉及其中一小部分：

- 通过调用sched_init()函数来初始化调度器。
- 通过build_all_zonelists()函数初始化memory zones。
- 通过调用page_alloc_init()和mem_init()函数来初始化Buddy子系统。
- IDT的最终初始化是通过调用trap_init()和init_IRQ()函数。
- 通过调用softirq_init()函数来初始化TASKLET_SOFTIRQ和HI_SOFTIRQ。
- 通过调用time_init()函数来初始化系统数据和时间。

- 通过调用kmem_cache_init()函数来初始化slab分配器。
- 通过调用calibrate_delay()来确认CPU速度。
- 通过调用kernel_thread()函数来创建1号内核线程。顺序的，这个内核线程创建了其它内核线程并执行/sbin/init程序。