1. 进程 0 创建进程 1 时，为进程 1 建立了自己的 task_struct、内核栈，第一个页表，分别位于物理内存 16MB 的顶端倒数第一页、第二页。请问，这个了页究竟占用的是谁的线性地址空间，内核、进程 0、进程 1、还是没有占用任何线性地址空间？说明理由并给出代码证据。

答：这两个页占用的是内核的线性地址空间，依据在 setup_paging（文件 head.s）中，

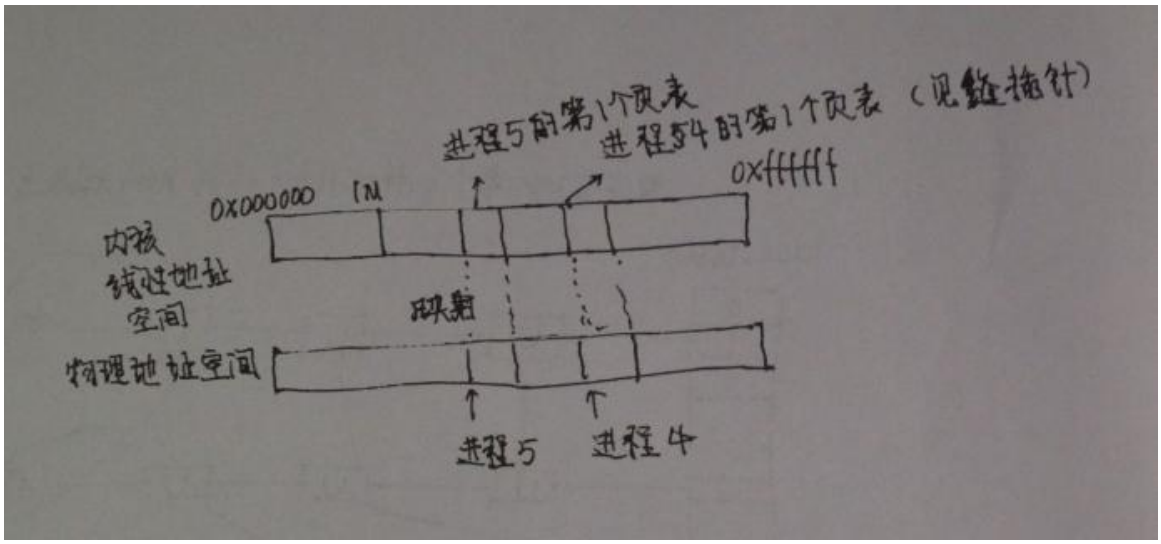
```
    movl $pg3+4092,%edi
    movl $0xfff007,%eax         /*  16Mb - 4096 + 7 (r/w user,p) */
    std
1:  stosl           /* fill pages backwards - more efficient :-) */
    subl $0x1000,%eax
```

上面的代码，指明了内核的线性地址空间为 0x000000 ~ 0xffffff（即前 16M），且线性地址与物理地址呈现一一对应的关系。为进程 1 分配的这两个页，在 16MB 的顶端倒数第一页、第二页，因此占用内核的线性地址空间。

进程 0 的线性地址空间是内存前 640KB，因为进程 0 的 LDT 中的 limit 属性限制了进程 0 能够访问的地址空间。进程 1 拷贝了进程 0 的页表（160 项），而这 160 个页表项即为内核第一个页表的前 160 项，指向的是物理内存前 640KB，因此无法访问到 16MB 的顶端倒数的两个页。

2. 假设：经过一段时间的运行，操作系统中已经有 5 个进程在运行，且内核分别为进程 4、进程 5 分别创建了第一个页表，这两个页表在谁的线性地址空间？用图表示这两个页表在线性地址空间和物理地址空间的映射关系。

答：这两个页表在内核的线性地址空间，映射关系如下：



3. 进程 0 开始创建进程 1，调用了 fork（），跟踪代码时我们发现，fork 代码执行了两次，第一次，跳过 init（）直接执行了 for(;;) pause()，第二次执行 fork 代码后，执行了 init（）。奇怪的是，我们在代码中并没有看见向后的 goto 语句，也没有看到循环语句，是什么原因导致反复执行？请说明理由，并给出代码证据。

答：进程 0 创建进程 1 采用了中断机制，在中断发生时由硬件将 ss，esp，eflags，cs，eip 的值压入了内核栈，其中 eip 的值指向了 int 0x80 的下一条指令。在执行 fork 时，通过 0x80 号系统调用，内核执行 copy_process 函数，为进程 1 准备其管理结构（task_struct），设置进

程 1 的线性地址空间及物理页面，其中设置了进程 1 的 TSS 中 eax 的值为 0，状态为 TASK_RUNNING，以及利用中断压栈的寄存器值设置进程 1 的 ss，esp，eflags，cs，eip。

copy_process:

```
    p->pid = last_pid;
    …
    p->tss.eip = eip;
    p->tss.eflags = eflags;
    p->tss.eax = 0;
    …
    p->tss.esp = esp;
    …
    p->tss.cs = cs & 0xffff;
    p->tss.ss = ss & 0xffff;
    …
    p->state = TASK_RUNNING;
    return last_pid;
```

函数 copy_process 的返回值是 last_pid，即进程 1 的 pid（pid 不为 0）。在 fork 返回到进程 0 后，进程 0 判断返回值非 0，因此执行代码

```
for(;;) pause();
```

在 sys_pause 函数中，内核设置了进程 0 的状态为 TASK_INTERRUPTIBLE，并执行了进程调度。由于只有进程 1 处于就绪态，因此调度执行进程 1 的指令。由于进程 1 在 TSS 中设置了 eip 等寄存器的值，因此从 int 0x80 的下一条指令开始执行，且设定返回 eax 的值作为 fork 的返回值（值为 0），因此进程 1 执行了 init 的函数。导致反复执行，主要是利用了两个系统调用 sys_fork 和 sys_pause 对进程状态的设置，以及利用了进程调度机制。

4. copy_process 函数的参数最后五项是：long eip,long cs,long eflags,long esp,long ss。查看栈结构确实有这五个参数，奇怪的是其他参数的压栈代码都能找得到，确找不到这五个参数的压栈代码，反汇编代码中也查不到，请解释原因。

答：这 5 个参数是在中断发生时，由硬件执行的压栈动作，因此在反汇编代码中是查不到的。利用硬件进行压栈，可以确保 eip 的值指向正确的指令，以使在中断返回后，程序能够继续执行。

5.用图表示下面的几种情况，并从代码中找到证据：

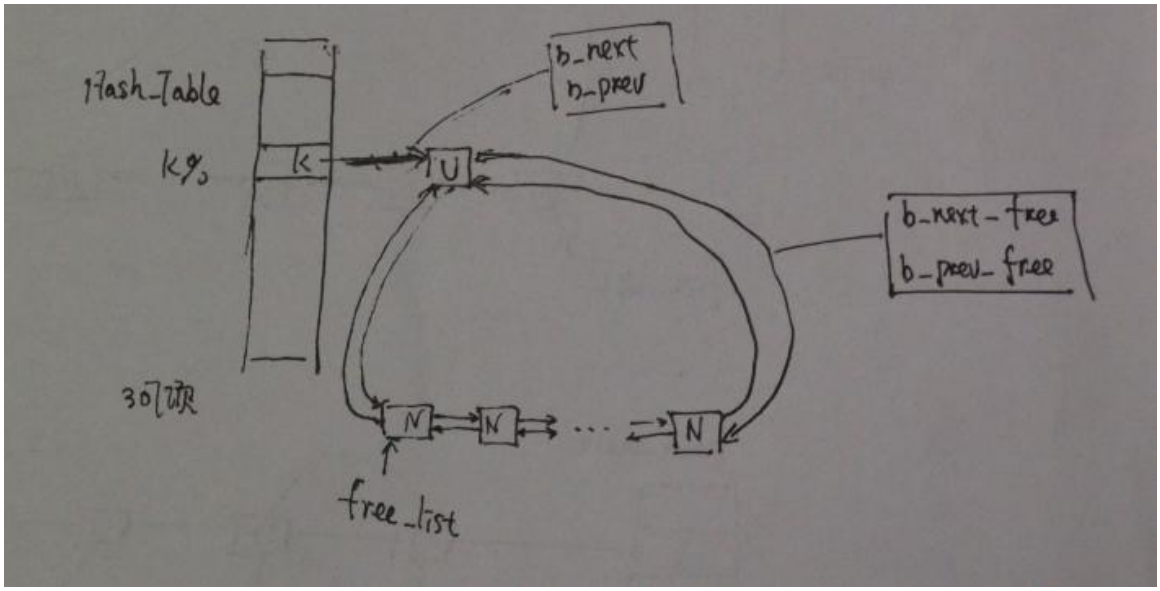
a、当进程获得第一个缓冲块的时候，hash 表的状态。

b、经过一段时间的运行，已经 2000 多个 buffer_head 挂到 hash_table 上时，hash 表（包括所有的 buffer_head）的整体状态。

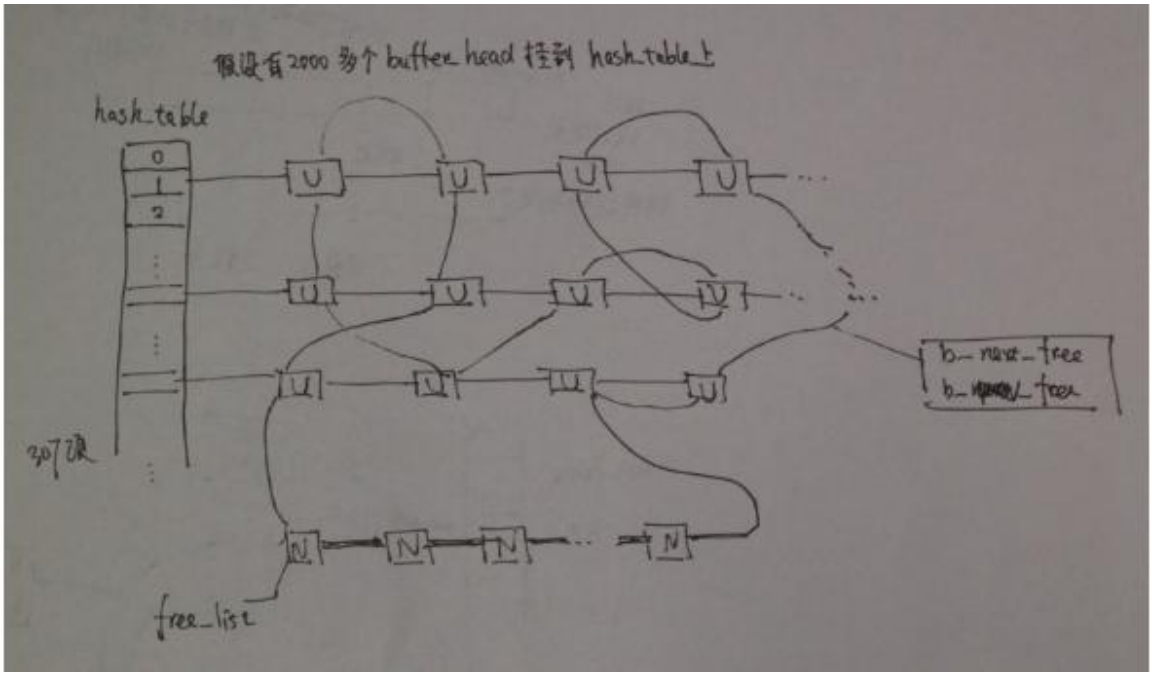
c、经过一段时间的运行，有的缓冲块已经没有进程使用了（空闲），这样的空闲缓冲块是否会从 hash_table 上脱钩？

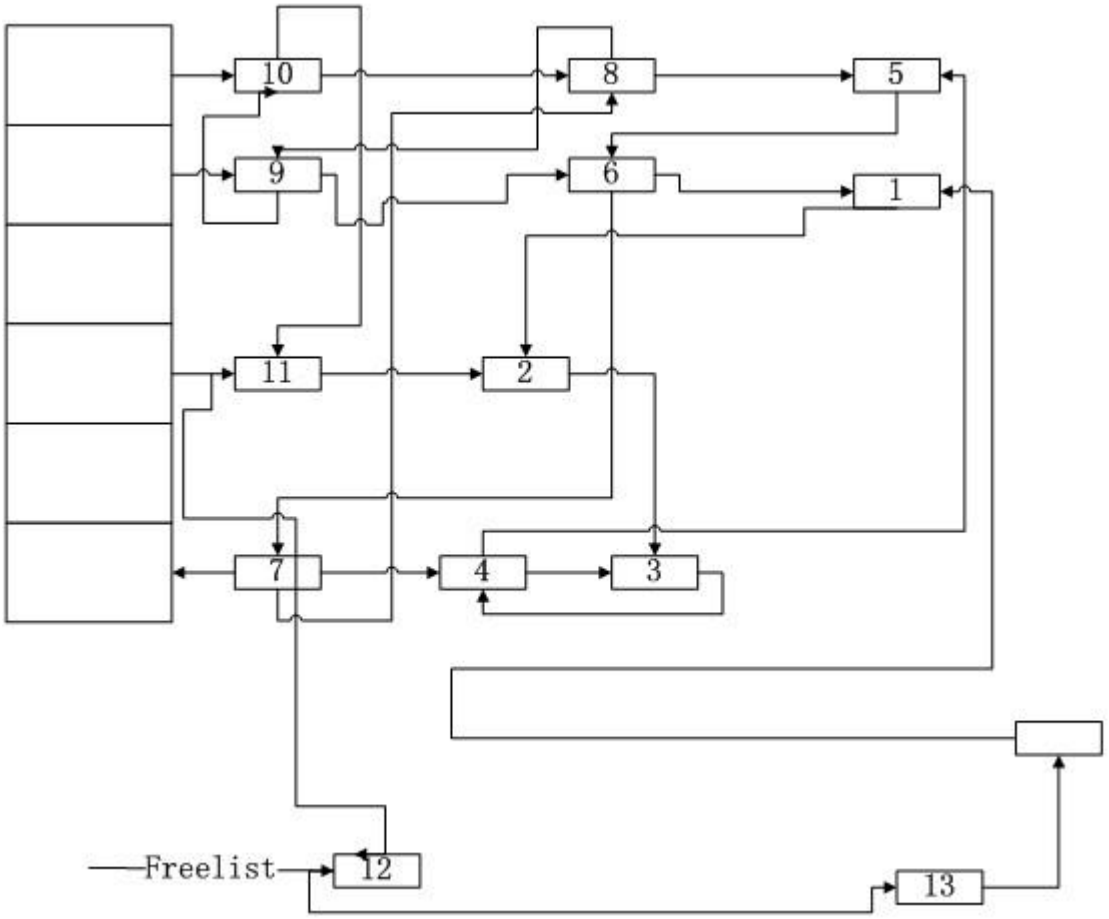
d、经过一段时间的运行，所有的 buffer_head 都挂到 hash_table 上了，这时，又有进程申请空闲缓冲块，将会发生什么？解释原因并找到代码证据。

答：a、


Hash_Table
K%
K
U
b_next
b_prev
b_next_free
b_prev_free
30万块
N
N
N
free_list

b、


假设有2000多个 buffer_head 挂到 hash_table 上
hash_table
0
1
2
U
b_next_free
30万块
N
free_list



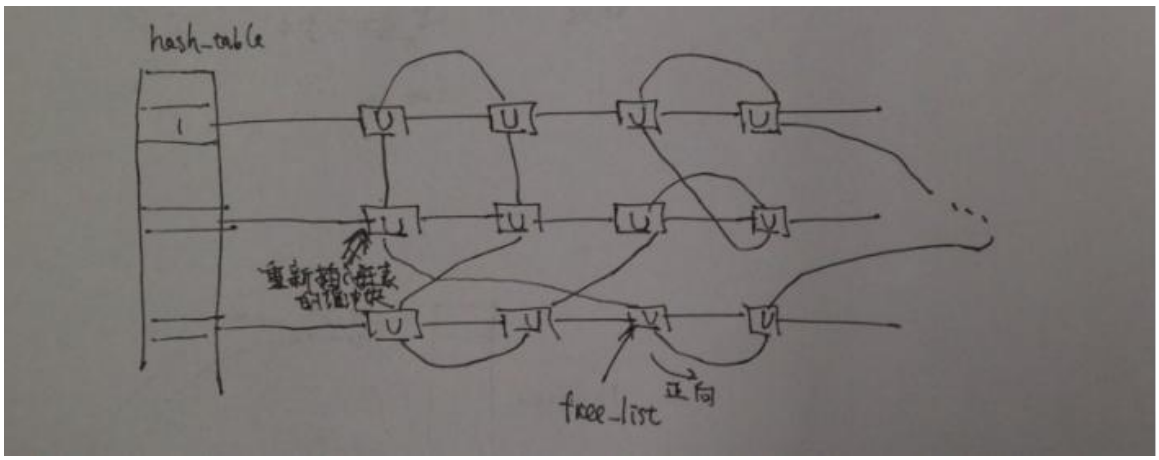
c、有的缓冲块没有进程使用，这样的空闲缓冲块不会从 hash_table 上脱钩。在 brelse 函数中，也只是将引用计数减小，没有对该缓冲块执行 remove 操作。由于硬盘读写开销一般比内存大几个数量级，因此该空闲缓冲块如果能够被再次访问到，对性能提升是有益的。

d、所有的 buffer_head 都挂到 hash_table 上了以后，如果进程再申请空闲缓冲块，那么需要从 free_list 指向的缓冲块开始，在缓冲块双向链表中寻找满足 b_count 为 0 的缓冲块，并且以 b_dirt, b_lock 作为度量尽量找寻未经 BADNESS(bh)值较低的缓冲块。如该缓冲块 b_dirt 位为 1，则应同步该缓冲块到硬盘。如该缓冲块 b_lock 位为 1 则应执行 wait_on_buffer。在最后获得了缓冲块后，执行 remove_from_queues，使得该缓冲块从 hash_table 和 free_list 链表中脱钩。再设置了 block 后，重新 insert_into_queues(bh)，即将该缓冲块放置到 free_list 双向链表的尾端（亦为 free_list 之前），以及插入 hash_table 中。代码见 getblk 函数，其中

```
bh->b_count=1;
bh->b_dirt=0;
bh->b_uptodate=0;
remove_from_queues(bh);
bh->b_dev=dev;
bh->b_blocknr=block;
insert_into_queues(bh);
```

上述代码为放置新缓冲块的部分。

图如下：



6. rd_load（）执行完之后，虚拟盘已经成为可用的块设备，并成为根设备。在向虚拟盘中 copy 任何数据之前，虚拟盘中是否有引导块、超级块、i 节点位图、逻辑块位图、i 节点、逻辑块？请解释其中的道理，并给出代码证据。

答：虚拟盘中没有引导块、超级块、i 节点位图、逻辑块位图、i 节点、逻辑块。在 rd_load() 函数中的 memcpy(cp,bh->b_data,BLOCK_SIZE)执行以前，对虚拟盘的操作仅限于为虚拟盘分配 2M 的内存空间，并将虚拟盘的所有内存区域初始化为 0。所以虚拟盘中并没有引导块等数据。

```
rd_start = (char *) mem_start;
        rd_length = length;
        cp = rd_start;
        for (i=0; i < length; i++)
                *cp++ = '\0';
```

Error Version: 虚拟盘中有引导块、超级块、i 节点位图、逻辑块位图、i 节点、逻辑块。rd_load() 是虚拟盘根文件加载函数。在系统初始化阶段，该函数被用于尝试从启动引导盘上指定的磁盘块位置开始处把一个根文件系统加载到虚拟盘中。在函数中，这个起始磁盘块位置被定为 256。虚拟盘成为根设备，蕴含了根文件系统被正常加载，因此虚拟盘中已有各种数据。证据在于函数 rd_load()中，读取了根文件系统（虚拟盘映像）的超级块，并对该超级块分析，之后加载了根文件系统所有数据块。

```
        if (nblocks > 2)
                bh = breada(ROOT_DEV, block, block+1, block+2, -1);
        else
                bh = bread(ROOT_DEV, block);
        … …
        (void) memcpy(cp, bh->b_data, BLOCK_SIZE);
```

7.在虚拟盘被设置为根设备之前，操作系统的根设备是软盘（包括软驱），请说明设置软盘为根设备的技术路线，并给出代码证据。

（提示：注意 bootsect.s 的 249 行，508 这个数值。

```
.org 508
root_dev:
```

.word ROOT_DEV

）

答：使用软盘作为启动盘，软盘的第一个扇区设置为引导扇区，

boot_flag:　　.word 0xAA55

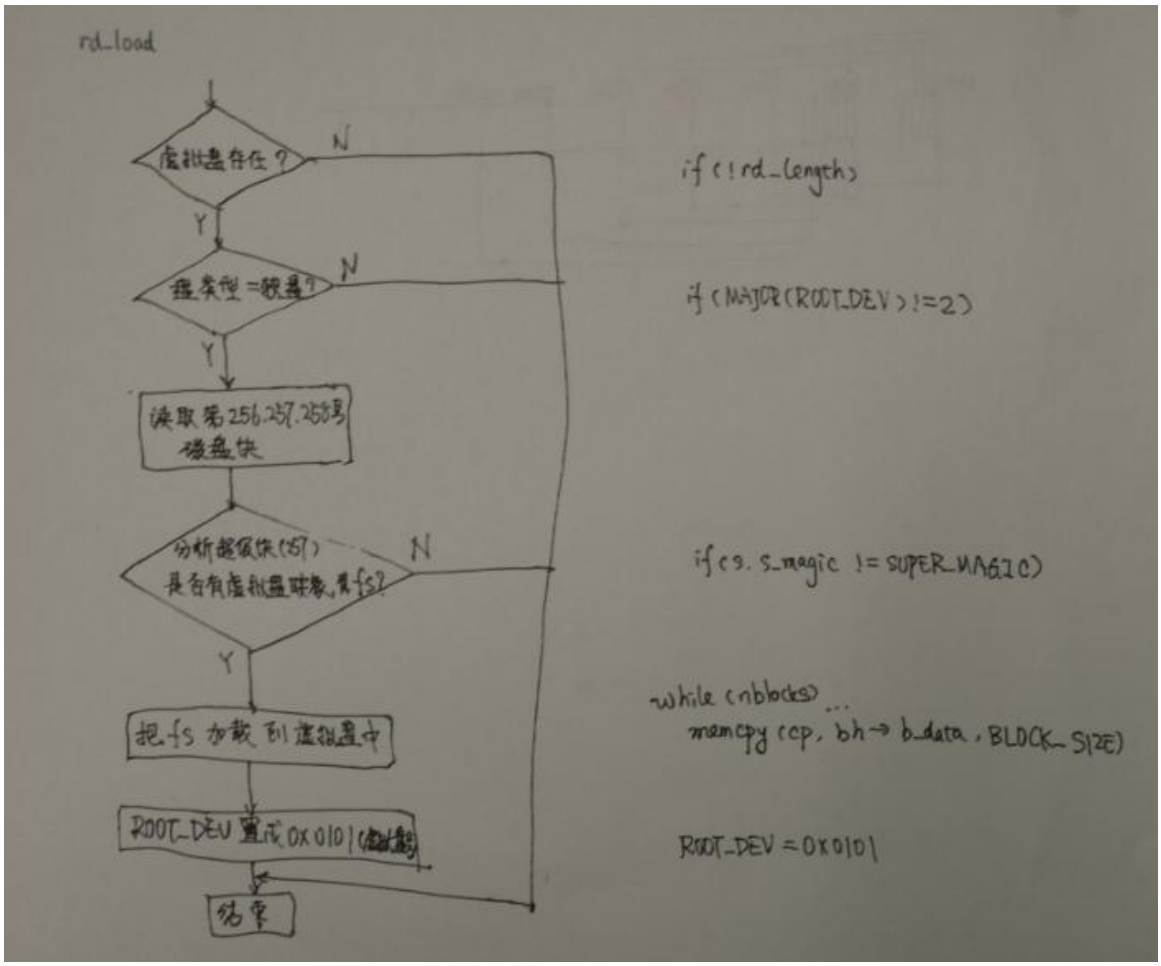
bootsect.s 在编译后文件的 508，509 处预留了空间，用于存放根设备号，0x306 是初始值。在主 Makefile 文件 ROOT_DEV=/dev/hd6。在把第一个扇区从软盘加载之后，第一个扇区又被复制到了 0x90000~0x901ff 的地方，因此根设备号被保存在地址 0x901fc 处。在 Main.c 中，

ROOT_DEV = ORIG_ROOT_DEV;

将根设备号保存在变量 ROOT_DEV 中。之后，对于软盘的读写，在 rd_load 中，MAJOR(ROOT_DEV)判断“盘类型=软盘”，如果是，则使用 ROOT_DEV 作为参数，从软盘指定的磁盘块加载根文件系统到虚拟盘上。如果加载成功，那么设定 ROOT_DEV 为虚拟盘(0x0101)。否则从退出 rd_load，系统举行从被的设备上执行根文件加载操作。

8. Linux0.11 是怎么将根设备从软盘更换为虚拟盘，并加载了根文件系统？用文字、图示表示，并给出代码证据。

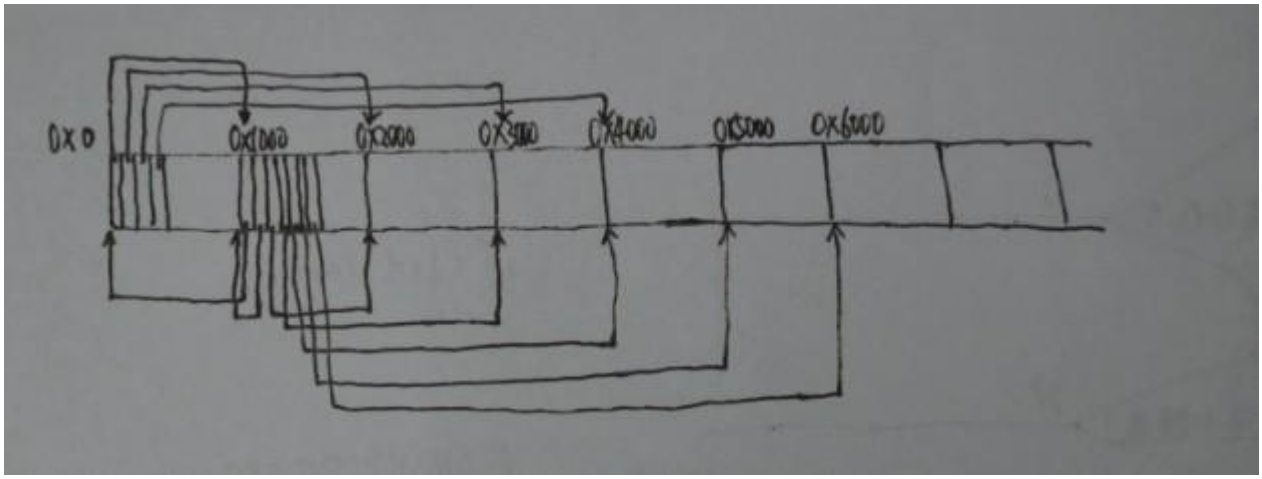
答：



赵炯《Linux 内核完全注释》Page 385 图 9-7

9.内核的线性地址空间是如何分页的？画出从 0x000000 开始的 7 个页（包括页目录表、页表所在页）的挂接关系图，就是页目录表的前四个页目录项、第一个个页表的前 7 个页表项指向什么位置？给出代码证据。

答：



代码证据 setup_paging:

```
    movl $pg0+7,_pg_dir
    movl $pg1+7,_pg_dir+4
    movl $pg2+7,_pg_dir+8
    movl $pg3+7,_pg_dir+12
```

其中_pg_dir 值为 0x0000。

```
    movl $pg3+4092,%edi
    movl $0xfff007,%eax          /*    16Mb - 4096 + 7 (r/w user,p) */
    std
1:  stosl           /* fill pages backwards - more efficient :-) */
    subl $0x1000,%eax
    jge 1b
```

另外，4 个页表挂满了 16MB 的内存，促成线性地址与物理地址的一一对应的关系。

10. 用文字和图说明中断描述符表是如何初始化的，可以举例说明（比如：set_trap_gate(0,&divide_error)），并给出代码证据。

答：如 set_trap_gate(0,&divide_error)，0 指定了 idt 中的偏移值，即第 0 项。中断描述符的类型 15(由高地址双字节中低 8-11 位指定)指定了这是一个 trap gate，用于异常出错处理，特权级为 0，addr 给出了中断处理程序的地址。

```
#define set_trap_gate(n,addr) \
    _set_gate(&idt[n],15,0,addr)

#define _set_gate(gate_addr,type,dpl,addr) \
__asm__ ("movw %%dx,%%ax\n\t" \
    "movw %0,%%dx\n\t" \
    "movl %%eax,%1\n\t" \
    "movl %%edx,%2" \
    : \
    : "i" ((short) (0x8000+(dpl<<13)+(type<<8))), \
```

```
"o" (*((char *) (gate_addr))), \
"o" (*(4+(char *) (gate_addr))), \
"d" ((char *) (addr)),"a" (0x00080000))
```

中断初始化，初始化了 0-16，45,39 号异常处理，其余的初始化为保留。

11.为什么计算机启动最开始的时候执行的是 BIOS 代码而不是操作系统自身的代码？

答：通常我们用 c 语言写的用户程序，在执行的时候总是在操作系统的平台上，即操作系统为应用程序创建进程并把应用程序的可执行代码加载到内存。计算机启动的时候，操作系统并没有在内存中，我们首先要把操作系统加载到内存，而这个工作最开始的部分，就是由 bios 程序来实现的。所以计算机启动最开始执行的是 bios 代码

12.为什么 BIOS 只加载了一个扇区，后续扇区却是由 bootsect 代码加载？为什么 BIOS 没有把所有需要加载的扇区都加载？

答：对 BIOS 而言，“约定”在接到启动操作系统的命令后，“定位识别”只从启动扇区把代码加载到 0x7c00 这个位置。后续扇区则由 bootsect 代码加载，这些代码由编写系统的用户负责，与 BIOS 无关。这样构建的好处是站在整个体系的高度，统一设计和统一安排，简单而有效。BIOS 和操作系统的开发都可以遵循这一约定，灵活地进行各自的设计。例如，BIOS 可以不用知道内核镜像的大小以及其在软盘的分布等等信息，减轻了 BIOS 程序的复杂度，降低了硬件上的开销。而操作系统的开发也可以按照自己的意愿，内存的规划，等等都更为灵活。

Reference:p40 and p6

13.为什么 BIOS 把 bootsect 加载到 0x07c00，而不是 0x00000？加载后又马上挪到 0x90000 处，是何道理？为什么不一次加载到位？

答：

第一问：

① "0x7C00" First appeared in 首次出现在 IBM PC 5150 ROM BIOS INT 19h handler（中断处理程序的地址），IBM PC 5150 BIOS Developer Team 决定使用这个地址的。

② "0x7C00"这个数字属于 BIOS 的规范范畴的

③ "0x7C00 = 32KiB - 1024B" 原因在于操作系统的需求和 CPU 内存布局

```
+-------------------- 0x0
| Interrupts vectors
+-------------------- 0x400
| BIOS data area
+-------------------- 0x5??
| OS load area
+-------------------- 0x7C00
| Boot sector
+-------------------- 0x7E00
```

| Boot data/stack

+-------------------- 0x7FFF

| (not used)

+-------------------- (...)

Reference: http://biancheng.dnbcw.info/linux/309367.html

第二问：

BIOS 把 0x7c00 的一个扇区挪到 0x90000，原因：

首先，在启动扇区中有一些数据，将会被内核利用到。如第 508、509 字节处的 ROOT_DEV。

其次，依据系统对内存的规划，内核终会占用 0x0000 其实的空间，因此 0x7c00 可能会被覆盖。

将该扇区挪到 0x90000，在 setup.s 中，获取一些硬件数据保存在 0x90000~0x901ff 处，可以对一些后面内核将要利用的数据，集中保存和管理。

14.bootsect、setup、head 程序之间是怎么衔接的？给出代码证据。

答：bootsect 在加载了 setup 段，加载内核，和检查 root_device 之后，执行

```
jmpi 0,SETUPSEG
```

由于 bootsect 将 setup 段加载到了 SETUPSEG:0 的地方，因此在实模式下，该指令跳转到了 setup 段的第一条指令。

setup 执行了之后，内核被移到了 0x0000 处，系统进入了保护模式，加载了 idt 和 gdt，

```
lidt  idt_48
lgdt  gdt_48
```

其中 gdt 中的第 2 项和第 3 项，分别初始化为代码段和数据段，其基地址为 0x0000，即内核被移动到的位置。开启保护模式后，执行

```
jmpi 0,8        ! jmp offset 0 of segment 8 (cs)
```

根据保护模式的机制，该指令执行后跳转到以 gdt 第 2 项中的 base_addr 为基地址，以 0 为偏移量的地方。由于 head 放置在内核的头部，因此程序跳转到 head 中执行。

15.setup 程序里的 cli 是为了什么？

答：因为在 setup 中，将 bios 中断向量表覆盖掉了，此时如果产生中断，会发生不可预知的错误，所以要禁示中断。

16.setup 程序的最后是 jmpi 0,8 为什么这个 8 不能简单的当作阿拉伯数字 8 看待？

答：这里 8 要看成二进制 1000，最后两位 00 表示内核特权级，第三位 0 表示 GDT 表，第四位 1 表示所选的表（在此就是 GDT 表）的 1 项来确定代码段的段基址和段限长等信息。这样，我们可以得到代码是从段基址 0x00000000、偏移为 0 处开始执行的，即 head 的开始位置。注意到已经开启了保护模式的机制，所以这里的 8 不能简单的当成阿拉伯数字 8 来看待。

17.打开 A20 和打开 pe 究竟是什么关系，保护模式不就是 32 位的吗？为什么还要打开 A20？有必要吗？

答：是相互独立的关系。

A20 是 cpu 的第 21 位地址线，A20 未打开的时候，实模式中 cs：ip 最大寻址为 1MB+64KB，而第 21 根地址线被强制为 0，所以相当于 cpu“回滚”到内存地址起始处寻址。当打开 A20 的

时候，实模式下 cpu 可以寻址到 1MB 以上的高端内存区。
A20 未打开时，如果打开 pe，则 cpu 进入保护模式，但是可以访问的内存只能是奇数 1M 段，即 0-1M,2M-3M,4-5M 等。A20 被打开后，如果打开 pe，则可以访问的内存是连续的。所以进入保护模式也有必要打开 A20
http://hi.baidu.com/%BF%E1%CE%D2%D2%BB%B4%FA/blog/item/c2d79b091f8a49dd3ac763d0.html
http://blog.csdn.net/yunsongice/article/details/6110648

18.Linux 是用 C 语言写的，为什么没有从 main 还是开始，而是先运行 3 个汇编程序，道理何在？
答：通常用 C 语言编写的程序都是用户应用程序，这类程序的执行必须在操作系统上执行，也就是说要由操作系统为应用程序创建进程，并把应用程序的可执行代码从硬盘加载到内存。

而对于操作系统，在刚加载的时候，计算机刚刚加电，只有个 BIOS 程序在运行，而且此时计算机处于 16 位实模式状态下，通过 BIOS 自身代码形成的 16 位中断向量表及相关的 16 位中断服务程序，将操作系统在软盘上的第一个扇区（512 字节）的代码加载到内存。从第二扇区开始，则由第一个扇区中的代码来完成后续代码的加载工作。

加载工作完成后，需要执行 3 个汇编程序，打开 A20，打开 PE 和 PG，建立 IDT、GDT 等等。Linux 0.11 是一个 32 位的实时多任务的现代操作系统，main 函数要执行的是 32 位的代码。开机时的 16 位实模式与 main 函数执行需要的 32 位保护模式之间有很大的差距，这个差距需要由 3 个汇编程序来填补。其中 bootsect 负责加载，setup 与 head 则负责获取硬件参数，准备 idt,gdt,开启 A20，PE,PG，废弃旧的 16 位中断响应机制，建立新的 32 为 IDT，设置分页机制等。这些工作做完后，计算机处在了 32 位的保护模式状态了，调用 main 的条件就算准备完毕。

Reference: (page 40)

19.为什么不用 call，而是用 ret“调用”main 函数？画出调用路线图，给出代码证据。
答：CALL 指令会将 EIP 的值自动压栈，保护返回现场，然后执行被掉函数的程序。等到执行那个被调函数的 ret 指令时，自动出栈给 EIP 并还原现场，继续执行那个 CALL 的下一行指令。但对于操作系统的 main 来说，如果 CALL 调用操作系统的 main 函数，那么 ret 时返回给谁？操作系统是最底层的系统，所以要达到既调用 main，又不需返回，就不采用 call 而是选择了 ret“调用”了。
调用路线图：见 Page 39 图 1-44.（自己画去:-）

代码证据：

```
after_page_tables:
        …
        pushl $L6       # return address for main, if it decides to.
        pushl $_main
        jmp setup_paging
…
setup_paging:
        …
```

ret

20.保护模式的“保护”体现在哪里？

答：打开了保护模式后，CPU 的寻址模式发生了变化。需要依赖于 GDT 的解析，获取代码或数据段的基址。

从 GDT 可以看出，保护模式除了段基址外，还有段限长，这样相当于增加了一个段位寄存器。既有效地防止了对代码或数据段的覆盖，又防止了代码段自身的访问超限，明显增强了保护作用。

而保护模式中的特权级，则对操作系统的“主奴机制”影响深远。Intel 从硬件上禁止低特权级代码段使用一些关键性指令，Intel 还提供了机会允许操作系统设计者通过一些特权级的设置，禁止用户进程使用 cli、sti 等对掌控局面至关重要的指令。有了这些基础，操作系统可以把内核设计成最高特权级，把用户进程设计成最低特权级。这样，操作系统可以访问 GDT、LDT、TR，而 GDT、LDT 是逻辑地址形成线性地址的关键，因此操作系统可以掌控线性地址。物理地址是由内核将线性地址转换而成的，所以操作系统可以访问任何物理地址。而用户进程只能使用逻辑地址。总之，特权级的引入对于操作系统内核提供了强有力的保护。

REFERENCE： PAGE402

21.特权级的目的和意义是什么？为什么特权级是基于段的？

答：特权级

目的：在于保护高特权级的段，其中操作系统的内核处于最高的特权级。

意义：保护模式中的特权级，对操作系统的“主奴机制”影响深远。Intel 从硬件上禁止低特权级代码段使用一些关键性指令，Intel 还提供了机会允许操作系统设计者通过一些特权级的设置，禁止用户进程使用 cli、sti 等对掌控局面至关重要的指令。有了这些基础，操作系统可以把内核设计成最高特权级，把用户进程设计成最低特权级。这样，操作系统可以访问 GDT、LDT、TR，而 GDT、LDT 是逻辑地址形成线性地址的关键，因此操作系统可以掌控线性地址。物理地址是由内核将线性地址转换而成的，所以操作系统可以访问任何物理地址。而用户进程只能使用逻辑地址。总之，特权级的引入对于操作系统内核提供了强有力的保护。

在操作系统设计中，一个段一般实现的功能相对完整，可以把代码放在一个段，数据放在一个段，并通过段选择符（包括 CS、SS、DS、ES、FS 和 GS）获取段的基址和特权级等信息。特权级基于段，这样当段选择子具有不匹配的特权级时，按照特权级规则评判是否可以访问。特权级基于段，是结合了程序的特点和硬件实现的一种考虑。

REFERENCE: PAGE405

22.在 setup 程序里曾经设置过一次 gdt，为什么在 head 程序中将其废弃，又重新设置了一个？为什么折腾两次，而不是一次搞好？

答：在 setup 程序里的 GDT，其位置将来会在设计缓冲区时被覆盖。如果不改变位置，GDT 的内容将来肯定会被缓冲区覆盖掉，从而影响系统的运行。而将来，整个内存中唯一安全的地方就是 head 程序的位置。

其次，不能在执行 setup 时直接把 GDT 内容拷贝到 head 所在的位置。因为，如果先复制 GDT 的内容，后移动 system 模块，它就会被后者覆盖掉；如果先移动 system 模块，后复

制 GDT 的内容，它又会把 head 对应的程序覆盖掉，而这时 head 还没有执行。所以无论如何，都要重新建立 GDT。
REFERENCE: PAGE 30

23.在 head 程序执行结束的时候，在 idt 的前面有 184 个字节的 head 程序的剩余代码，剩余了什么？为什么要剩余？
答：在 idt 前面有 184 个字节的剩余代码，包含了 after_page_tables、ignore_int 和 setup_paging 代码段。其中 after_page_tables 往栈中压入了些参数，ignore_int 用做初始化中断时的中断处理函数，setup_paging 则是初始化分页。
剩余的原因：
after_page_tables 中压入了一些参数，为内核进入 main 函数的跳转做准备。为了谨慎起见，设计者在栈中压入了 L6，以使得系统可能出错时，返回到 L6 处执行。
ignore_int 为中断处理函数，使用 ignore_int 将 idt 全部初始化，因此如果中断开启后，可能使用了未设置的中断向量，那么将默认跳转到 ignore_int 处执行。这样做的好处是使得系统不会跳转到随机的地方执行错误的代码，所以 ignore_int 不能被覆盖。
setup_paging 用于分页，在该函数中对 0x0000 和 0x5000 的进行了初始化操作。该代码需要“剩余”用于跳转到 main，即执行”ret”指令。

24.进程 0 的 task_struct 在哪？具体内容是什么？给出代码证据。
答：进程 0 的 task_struct 位于内核数据区，即 task 结构的第 0 项 init_task。

```
struct task_struct * task[NR_TASKS] = {&(init_task.task), };
```

具体内容：包含了进程 0 的进程状态、进程 0 的 LDT、进程 0 的 TSS 等等。其中 ldt 设置了代码段和堆栈段的基址和限长(640KB)，而 TSS 则保存了各种寄存器的值，包括各个段选择符。具体值如下：

```
#define INIT_TASK \
/* state etc */ { 0,15,15, \
…
    { \
        {0,0}, \
/* ldt */  {0x9f,0xc0fa00}, \
        {0x9f,0xc0f200}, \
    }, \
/*tss*/    {0,PAGE_SIZE+(long)&init_task,0x10,0,0,0,0,(long)&pg_dir,\
     0,0,0,0,0,0,0,0, \
     0,0,0x17,0x17,0x17,0x17,0x17,0x17, \
     _LDT(0),0x80000000, \
        {} \
    }, \
```

25.在 system.h 里

```
#define _set_gate(gate_addr,type,dpl,addr) \
__asm__ ("movw %%dx,%%ax\n\t" \
    "movw %0,%%dx\n\t" \
```

```
        "movl %%eax,%1\n\t" \
        "movl %%edx,%2" \
        : \
        : "i" ((short) (0x8000+(dpl<<13)+(type<<8))), \
        "o" (*((char *) (gate_addr))), \
        "o" (*(4+(char *) (gate_addr))), \
        "d" ((char *) (addr)),"a" (0x00080000))

#define set_intr_gate(n,addr) \
        _set_gate(&idt[n],14,0,addr)

#define set_trap_gate(n,addr) \
        _set_gate(&idt[n],15,0,addr)

#define set_system_gate(n,addr) \
        _set_gate(&idt[n],15,3,addr)
```

这里中断门、陷阱门、系统调用都是通过_set_gate 设置的，用的是同一个嵌入汇编代码，比较明显的差别是 dpl 一个是 3，另外两个是 0，这是为什么？说明理由。

答：系统调用是操作系统专门为用户进程写的程序，所以其 dpl 应该为三特权级；而中断和陷阱则是给系统的处理程序，用户不能访问，所以特权级定为 0 特权级。

Version 2.

system_call 是整个操作系统中系统调用软中断的总入口，所有用户程序产生了系统调用软中断后，系统都是通过这个总入口进一步找到具体的系统调用函数的。而用户程序的特权级是 3，所以设置 dpl 为 3 以便允许用户程序调用。

而其余的两个异常处理与外设的中断服务程序都是不允许用户调用的，所以设置特权级为 0，即 dpl 为 0。

27.进程 0 fork 进程 1 之前，为什么先要调用 move_to_user_mode()？用的是什么方法？解释其中的道理。

答：在执行 move_to_user_mode()之前，执行的是特权级为 0 的内核程序。在执行之后，才真正的使用进程 0 的结构，包括进程 0 的 LDT 和 TSS 等，此时 TR 和 LDTR 已经加载了进程 0 的信息，系统运行在用户进程的模式下。而 LDT 规定了进程 0 的代码段和数据段的地址空间。

进程 0fork 进程 1，使用的是 0x80 号系统调用（中断机制），且采用了父子进程创建机制。系统的设计者，安排了让特权级为 0 的内核执行创建进程的主要过程，内核负有对进程进行管理的权限。而且由于内核可以访问到更多的内核管理结构，因此采用中断可以由低特权级跳转到高特权级，再由内核创建进程 1。而父子进程创建机制，进程 1 在创建的时候复制了进程 0 的资源（如 copy_page_tables），通过区分 fork 的返回值，使得父子进程执行不同的代码部分。

28.进程 0 创建进程 1 时调用 copy_process 函数，在其中直接、间接调用了两次 get_free_page 函数，在物理内存中获得了两个页，分别用作什么？是怎么设置的？给出代码证据。

答：第一个设置进程 1 的 tasks_truct，另外设置了进程 1 的堆栈段

sched.c

```
struct task_struct *current = &(init_task.task);
```

fork.c

```
p = (struct task_struct *) get_free_page();
…
*p = *current;
p->tss.esp0 = PAGE_SIZE + (long) p;
```

其中*current 即为进程 0 的 task 结构，在 copy_process 中，先复制进程 0 的 task_struct，然后再对其中的值进行修改。esp0 的设置，意味着设置该页末尾为进程 1 的堆栈的起始地址。

第二个为进程 1 的页表，在创建进程 1 执行 copy_process 中，执行 copy_mem(nr,p)时，内核为进程 1 拷贝了进程 0 的页表（160 项）。

copy_mem

```
…
if (copy_page_tables(old_data_base,new_data_base,data_limit)){
…
```

其中，copy_page_tables 内部

```
…
from_page_table = (unsigned long *) (0xfffff000 & *from_dir);
if (!(to_page_table = (unsigned long *) get_free_page()))
…
for ( ; nr-- > 0 ; from_page_table++,to_page_table++) {
…
}
```

获取了新的页，且从 from_page_table 将页表值拷贝到 to_page_table 处。

29.在 IA-32 中，有大约 20 多个指令是只能在 0 特权级下使用，其他的指令，比如 cli，并没有这个约定。奇怪的是，在 Linux0.11 中，在 3 特权级的进程代码并不能使用 cli 指令，会报特权级错误，这是为什么？请解释并给出代码证据。

答：根据 Intel Manual，cli 和 sti 指令与 CPL 和 EFLAGS[IOPL]有关。

CLI：如果 CPL 的权限高于等于 eflags 中的 IOPL 的权限，即数值上：cpl <= IOPL，则 IF 位清除为 0；否则它不受影响。EFLAGS 寄存器中的其他标志不受影响。

#GP(0) – 如果 CPL 大于（特权更小）当前程序或过程的 IOPL，产生保护模式异常。

由于在内核 IOPL 的值初始时为 0，且未经改变。进程 0 在 move_to_user_mode 中，继承了内核的 eflags。

move_to_user_mode()

```
…
"pushfl\n\t" \
…
```

"iret\n" \

而进程 1 再 copy_process 中，在进程的 TSS 中，设置了 eflags 中的 IOPL 位为 0。总之，通过设置 IOPL，可以限制 3 特权级的进程代码使用 cli 指令。

http://www.oldlinux.org/oldlinux/viewthread.php?tid=13148

http://scc.qibebt.cas.cn/docs/optimization/VTune(TM)%20User's%20Guide/mergedProjects/analyzer_ec/mergedProjects/reference_olh/instruct32_hh/vc32.htm

30.在 Linux 操作系统中大量的使用了中断、异常类的处理，为什么？究竟有什么好处？

答：CPU 是主机中关键的组成部分，进程在主机中的运算肯定离不开 CPU，而 CPU 在参与运算过程中免不了进行“异常处理”，这些异常处理都需要具体的服务程序来执行。这种 32 位中断服务体系是为适应一种被动相应中断信号的机制而建立的。这样 CPU 就可以把全部精力都放在为用户程序服务上，对于随时可能产生而又不可能时时都产生的中断信号，不用刻意去考虑，这就提高了操作系统的综合效率。以“被动相应”模式替代“主动轮询”模式来处理中断问题是现代操作系统之所以称之为“现代”的一个重要标志。

REFERENCE: PAGE 48