1 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？

计算机启动的时候，内存未初始化， CPU 不能直接从外设运行操作系统，所以必须将操作系统加载至内存中。而这个工作最开始的部分，BIOS 需要完成一些检测工作，和设置实模式下的中断向量表和服务程序，并将操作系统的引导扇区加载值 0x7C00 处，然后将跳转至 0x7C00。这些就是由 bios 程序来实现的。所以计算机启动最开始执行的是 bios 代码。

2.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？

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。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

3.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？

1）因为 BIOS 将从 0x00000 开始的 1KB 字节构建了了中断向量表，接着的 256KB 字节内存空间构建了 BIOS 数据区，所以不能把 bootsect 加载到 0x00000. 0X07c00 是 BIOS 设置的内存地址，不是 bootsect 能够决定的。

2）首先，在启动扇区中有一些数据，将会被内核利用到。

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。

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

1)bootsect 跳转到 setup 程序： jmpi 0,SETUPSEG;

这条语句跳转到 0x90200 处，即 setup 程序加载的位子，CS:IP 指向 setup 程序的第一条指令，意味着 setup 开始执行。

2)setup 跳转到 head 程序：CPU 工作模式首先转变为保护模式然后执行 jmpi 0,8

0 指的是段内偏移，8 是保护模式下的段选择符：01000，其中后两位表示内核特权级，第三位 0 代表 GDT，1 则表示 GDT 表中的第一项，即内核代码段，段基质为 0x0000000,而 head 程序地址就在这里，意味着 head 程序开始执行。

5.setup 程序里的 cli 是为了什么？

cli 为关中断，以为着程序在接下来的执行过程中，无论是否发生中断，系统都不再对此中断进行响应。

因为在 setup 中，需要将位于 0x10000 的内核程序复制到 0x0000 处，bios 中断向量表覆盖掉了，若此时如果产生中断，这将破坏原有的中断机制会发生不可预知的错误，所以要禁示中断。

6.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？

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。

7.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？

1、打开 A20 仅仅意味着 CPU 可以进行 32 位寻址，且最大寻址空间是 4GB。打开 PE 是进入保护模式。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。打开 A20 是打开 PE 的必要条件；而打开 A20 不一定非得打开 PE。

2、有必要。打开 PE 只是说明系统处于保护模式下，但若真正在保护模式下工作，必须打开 A20，实现 32 位寻址。

8.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？

main 函数运行在 32 位的保护模式下，但系统启动时默认为 16 位的实模式， 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。

9.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。（图在 P42）

call 指令会将 EIP 的值自动压栈，保护返回现场，然后执行被调函数的程序，等到执行被调函数的 ret 指令时，自动出栈给 EIP 并还原现场，继续执行 call 的下一条指令。然而对操作系统的 main 函数来说，如果用 call 调用 main 函数，那么 ret 时返回给谁呢？因为没有更底层的函数程序接收操作系统的返回。用 ret 实现的调用操作当然就不需要返回了，call 做的压栈和跳转动作需要手工编写代码。

```
after_page_tables:
    pushl $__main; //将 main 的地址压入栈，即 EIP
setup_paging:
    ret; //弹出 EIP，针对 EIP 指向的值继续执行，即 main 函数的入口地址。
```

10.保护模式的“保护”体现在哪里？

1）在 GDT、 LDT 及 IDT 中，均有自己界限，特权级等属性，这是对描述符所描述的对象的保护

2）在不同特权级间访问时，系统会对 CPL、 RPL、 DPL、 IOPL 等进行检验，对不同层级的程 序进行保护， 同还限制某些特殊指令的使用， 如 lgdt, lidt,cli 等

3）分页机制中 PDE 和 PTE 中的 R/W 和 U/S 等，提供了页级保护。分页机制将线性地址与物 理地址加以映射，提供了对物理地址的保护。

P438

11.特权级的目的和意义是什么？（为什么特权级是基于段的？注释：老师给的题目没有）

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。
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。

12.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？
原来的 GDT 位于 setup 中，将来此段内存会被缓冲区覆盖， 所以必须将 GDT 设置 head.s 所在 位置。
如果先将 GDT 设置在 head 所在区域，然后移动 system 模块，则 GDT 会被覆盖掉，如果先移 动 system 再复制 GDT，则 head.s 对应的程序会被覆盖掉，所以必须重建 GDT。若先移动 system 至 0x0000 再将 GDT 复制到 0x5cb8~0x64b8 处，虽可以实现， 但由于 setup.s 与 head.s 连接时不在同一文件， setup 无法直接获取 head 中的 gdt 的偏移量，需事先写入， 这会使设计失去一般性，给程序编写带来很大不便。

13.用户进程自己设计一套 LDT 表，并与 GDT 挂接，是否可行，为什么？(和 28 题重复)
不可行。首先，用户进程不可以设置 GDT、LDT，因为 Linux0.11 将 GDT、LDT 这两个数据结构设置在内核数据区，是 0 特权级的，只有 0 特权级的额代码才能修改设置 GDT、LDT；而且，用户也不可以在自己的数据段按照自己的意愿重新做一套 GDT、LDT，如果仅仅是形式上做一套和 GDT、LDT 一样的数据结构是可以的，但是真正起作用的 GDT、LDT 是 CPU 硬件认定的，这两个数据结构的首地址必须挂载在 CPU 中的 GDTR、LDTR 上，运行时 CPU 只认 GDTR 和 LDTR 指向的数据结构，其他数据结构就算起名字叫 GDT、LDT，CPU 也一概不认；另外，用户进程也不能将自己制作的 GDT、LDT 挂接到 GDRT、LDRT 上，因为对 GDTR 和 LDTR 的设置只能在 0 特权级别下执行,3 特权级别下无法把这套结构挂接在 CR3 上。

14 进程 0 的 task_struct、内核栈、用户栈在哪？给出代码证据。

这里是另一个同学做的，我之后补上

15.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（直接从物理地址分配）？说明理由并给出代码证据。
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，

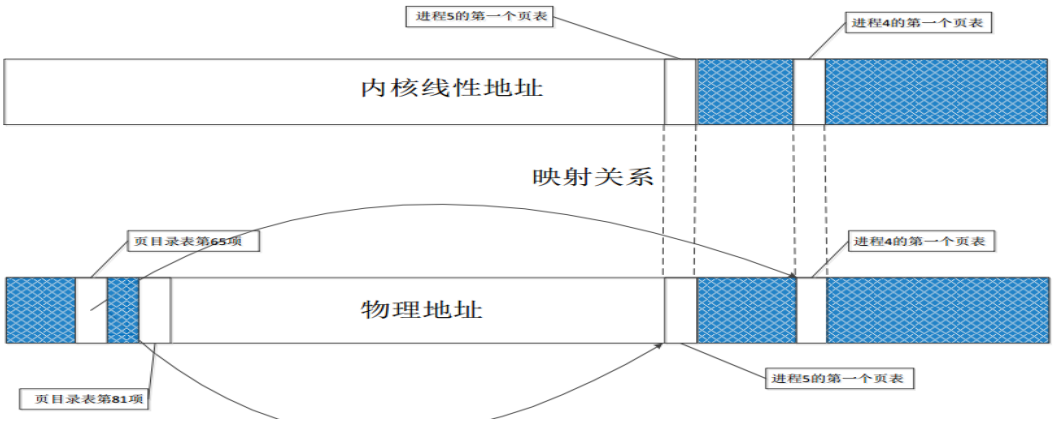
```
    movl $pg3+4092,%edi
    movl $0xfff007,%eax    /*    16Mb -4096 + 7 (r/w user,p) */
    std
1:  stosl/* fill pages backwards -more efficient :-) */
      subl $0x1000,%eax
```

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。

16.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。

这两个页面均占用内核的线性空间。



17.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。

首先在 copy_process（）函数中，对进程 1 做个性化调整设置，调整 tss 的数据。

Int copy_process（int nr, long ebp,…）

P92

然后再执行到如下代码：

＃define switch_to()……

ljmp….

p196

程序在执行到“ljmp %0/n/t”这一行，ljmp 通过 cpu 任务们机制自动将进程 1 的 tss 值恢复给 cpu,自然也将其中 tss.eip 恢复给 cpu,现在 cpu 指向 fork 的 if(_res >=0)这一行。

而此时的_res 值就是进程 1 中 tss 的 eax 的值，这个值在前面被写死为 0，即 p->tss.eax=0;

所以直行到 return (type) _res 这一行返回值为 0.

```
Voidmain（void）
 {
if（！ fork（））
 {
init（）；
}
}
```

返回后，执行到 main 函数中 if(!fork())这一行，!0 值为真，调用 init（）函数。

18.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。

copy_process 执行时因为进程调用了 fork 函数，会导致中断，中断使 CPU 硬件自动将 SS、ESP、EFLAGS、CS、EIP 这几个寄存器的值按照顺序压入 进程 0 内核栈，又因为函数专递参数是使用栈的，所以刚好可以做为 copy_process 的最后五项参数。

19.为什么 static inline _syscall0(type,name)中需要加上关键字 inline？

因为_syscall0(int,fork)展开是一个真函数，普通真函数调用事需要将 eip 入栈，返回时需要讲 eip 出栈。inline 是内联函数，它将标明为 inline 的函数代码放在符号表中，而此处的 fork 函数需要调用两次，加上 inline 后先进行词法分析、语法分析正确后就地展开函数，不需要有普通函数的 call\ret 等指令，也不需要保持栈的 eip，效率很高。若不加上 inline，第一次调用 fork 结束时将 eip 出栈，第二次调用返回的 eip 出栈值将是一个错误值。

20.根据代码详细说明 copy_process 函数的所有参数是如何形成的？

long eip, long cs, long eflags, long esp, long ss；这五个参数是中断使 CPU 自动压栈的。

long ebx, long ecx, long edx, long fs, long es, long ds 为__system_call 压进栈的参数。

long none 为__system_call 调用__sys_fork 压进栈 EIP 的值。

Int nr, long ebp, long edi, long esi, long gs,为__system_call 压进栈的值。

21.根据代码详细分析，进程 0 如何根据调度第一次切换到进程 1 的。（P103-107）

1.进程 0 通过 fork 函数创建进程 1，使其处在就绪态。

2.进程 0 调用 pause 函数。pause 函数通过 int 0x80 中断，映射到 sys_pause 函数，将自身设为可中断等待状态，调用 schedule 函数。

3.schedule 函数分析到当前有必要进行进程调度，第一次遍历进程，只要地址指针不为为空，就要针对处理。第二次遍历所有进程，比较进程的状态和时间骗，找出处在就绪态且 counter 最大的进程，此时只有进程 0 和 1，且进程 0 是可中断等待状态，只有进程 1 是就绪态，所以切换到进程 1 去执行。

22.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。

head.s 再 setup_paging 开始创建分页机制。将页目录表和 4 个页表放到物理内存的起始位置，从内存起始位置开始的 5 个页空间内容全部清零（每页 4kb），然后设置页目录表的前 4 项，使之分别指向 4 个页表。然后开始从高地址向低地址方向填写 4 个页表，依次指向内存从高地址向低地址方向的各个页面。即将第 4 个页表的最后一项（pg3+4092 指向的位置）指向寻址范围的最后一个页面。即从 0xFFF000 开始的 4kb 大小的内存空间。将第 4 个页表的倒数第二个页表项（pg3-4+4092）指向倒数第二个页面，即 0xFFF000-0x1000 开始的 4KB 字节的内存空间，依此类推。

Head.s 中：（P39）

```
 setup_paging:
movl $1024*5,%ecx  /* 5 pages - pg_dir+4 page tables */
```

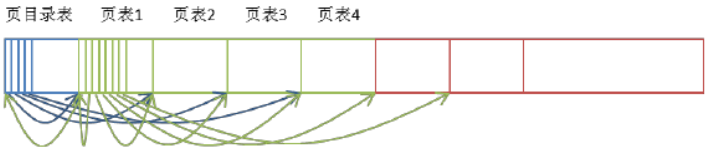
```
xorl %eax,%eax
xorl %edi,%edi  /* pg_dir is at 0x000 */
cld;rep;stosl
 movl $pg0+7,pg_dir  /* set present bit/user r/w */
movl $pg1+7,pg_dir+4  /*  --------- " " --------- */
movl $pg2+7,pg_dir+8  /*  --------- " " --------- */
movl $pg3+7,pg_dir+12  /*  --------- " " --------- */
```

_pg_dir 用于表示内核分页机制完成后的内核起始位置，也就是物理内存的起始位置 0x000000，以上四句完成页目录表的前四项与页表 1，2,3,4 的挂接

```
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
```

完成页表项与页面的挂接，是从高地址向低地址方向完成挂接的，16M 内存全部完成挂接（注意页表从 0 开始，页表 0-页表 3）

图见 P39



23. 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。

对中断描述符表的初始化，就是将异常处理一类的中断服务程序与中断描述符表进行挂接。以 set_trap_gate(0,&divide_error)为例，0 就表示该中断函数的地址挂接在中断描述符表的第 0 项位置处，而&devide_error 就是该异常处理函数的地址，对 set_trap_gate(0,&divide_error)进行宏展开后得到

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

之后执行如下代码：

```
#define _set_gate(&idt[0],15,0,&devide_error)(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+(0<<13)+(15<<8))), \
      "o" (*((char *) (&idt[0]))), \
      "o" (*(4+(char *) (&idt[0]))), \
      "d" ((char *) (&devide_error)),"a" (0x00080000))
```

%0=0x8f00，%1 指向 idt[0]的起始地址，%2 指向四个字节之后的地址处。

#1、将地址&devide_error 放在 EAX 的低两个字节，EAX 的高两字节不变。#2 把 0x8f00 放入 EDX 的低两字节，高两字节保持不变。#3、把 EAX 放在%1 所指的地址处，占四字节。#4、将 EDX 放在%2 所指的地址处，占四字节。

24.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。（P78-79）

因为在 Linux-0.11 中，除进程 0 之外，所有进程都是由一个已有进程在用户态下完成创建的。但是此时进程 0 还处于内核态，因此要调用 move_to_user_mode()函数，模仿中断返回的方式，实现进程 0 的特权级从内核态转化为用户态。又因为在 Linux-0.11 中，转换特权级时采用中断和中断返回的方式，调用系统中断实现从 3 到 0 的特权级转换，中断返回时转换为 3 特权级。因此，进程 0 从 0 特权级到 3 特权级转换时采用的是模仿中断返回。

25.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。（P89 91 92P97-98 ）

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。

26.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。

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。

27.根据代码详细分析操作系统是如何获得一个空闲页的。（P89-90）
通过逆向扫描页表位图 mem_map，并由第一空页的下标左移 12 位加 LOW_MEM 得到该页的物理地址， 位于 16M 内存末端。 代码如下（ get_free_page）
```
unsigned long get_free_page(void)
64 {
65 register unsigned long __res asm("ax");
66
67 __asm__("std ; repne ; scasb\n\t" //反向扫描串,al(0)与 di 不等则重复
68 "jne 1f\n\t"   //找不到空闲页跳转 1
69 "movb $1,1(%%edi)\n\t"   //将 1 付给 edi+1 的位置，在 mem_map 中将找到 0 的项引用计
----------------------------------------数置为 1
70 "sall $12,%%ecx\n\t"     //ecx 算数左移 12 位，页的相对地址
71 "addl %2,%%ecx\n\t"       //LOW MEN +ecx 页物理地址
72 "movl %%ecx,%%edx\n\t"
73 "movl $1024,%%ecx\n\t"
74 "leal 4092(%%edx),%%edi\n\t"         //将 edx+4kb 的有效地址赋给 edi
75 "rep ; stosl\n\t"              //将 eax 赋给 edi 指向的地址，目的是页面清零。
76 " movl %%edx,%%eax\n"
77 "1: cld"
78 :"=a" (__res)
79 :"0" (0),"i" (LOW_MEM),"c" (PAGING_PAGES),
80 "D" (mem_map+PAGING_PAGES-1)                    //edx,mem_map[]的嘴鸥一个元素
81 );
82 return __res;
83 }
```

28、用户进程自己设计一套 LDT 表，并与 GDT 挂接，是否可行，为什么？（P259）
不可行。首先，用户进程不可以设置 GDT、LDT，因为 Linux0.11 将 GDT、LDT 这两个数据结构设置在内核数据区，是 0 特权级的，只有 0 特权级的额代码才能修改设置 GDT、LDT；而且，用户也不可以在自己的数据段按照自己的意愿重新做一套 GDT、LDT，如果仅仅是形式上做一套和 GDT、LDT 一样的数据结构是可以的，但是真正起作用的 GDT、LDT 是 CPU 硬件认定的，这两个数据结构的首地址必须挂载在 CPU 中的 GDTR、LDTR 上，运行时 CPU 只认 GDTR 和 LDTR 指向的数据结构，其他数据结构就算起名字叫 GDT、LDT，

CPU 也一概不认；另外，用户进程也不能将自己制作的 GDT、LDT 挂接到 GDRT、LDRT 上，因为对 GDTR 和 LDTR 的设置只能在 0 特权级别下执行,3 特权级别下无法把这套结构挂接在 CR3 上。

29、保护模式下，线性地址到物理地址的转化过程是什么？（P260-261）

在保护模式下，先行地址到物理地址的转化是通过内存分页管理机制实现的。其基本原理是将整个线性和物理内存区域划分为 4K 大小的内存页面，系统以页为单位进行分配和回收。每个线性地址为 32 位，MMU 按照 10-10-12 的长度来识别线性地址的值。CR3 中存储着页目录表的基址，线性地址的前十位表示也目录表中的页目录项，由此得到所在的页表地址。21~12 位记录了页表中的页表项位置，由此得到页的位置，最后 12 位表示页内偏移。

30、详细分析进程调度的全过程。考虑所有可能（signal、alarm 除外）

首先在 task 数据（进程槽）中，从后往前进行遍历，寻找进程槽中，进程状态为“就緒态”且时间片最大的进程作为下一个要执行的进程。通过调用 switch_to()函数跳转到指定进程。在此过程中，如果发现存在状态为“就緒态”的进程，但这些进程都没有时间片了，则会从后往前遍历进程槽为所有进程重新分配时间片（不仅仅是“就緒态”的进程）。然后再重新执行以上步骤，寻找进程槽中，进程状态为“就緒态”且时间片最大的进程作为下一个要执行的进程。如果在遍历的过程中，发现没有进程处于“就绪态”，则会调用 switch_to()函数跳转到进程 0。

在 switch_to 函数中，如果要跳转的目标进程就是当前进程，则不发生跳转。否则，保存当前进程信息，长跳转到目标进程。

31、内核和普通用户进程并不在一个线性地址空间内，为什么仍然能够访问普通用户进程的页面？（P272）

内核的线性地址空间和用户进程不一样，内核是不能通过跨越线性地址访问进程的，但由于早就占有了所有的页面，而且特权级是 0，所以内核执行时，可以对所有的内容进行改动，“等价于”可以操作所有进程所在的页面。

32、wait_on_buffer 函数中为什么不用 if（）而是用 while（）？

因为可能存在一种情况是，很多进程都在等待一个缓冲块。在缓冲块同步完毕，唤醒各等待进程到轮转到某一进程的过程中，很有可能此时的缓冲块又被其它进程所占用，并被加上了锁。此时如果用 if()，则此进程会从之前被挂起的地方继续执行，不会再判断是否缓冲块已被占用而直接使用，就会出现错误；而如果用 while()，则此进程会再次确认缓冲块是否已被占用，在确认未被占用后，才会使用，这样就不会发生之前那样的错误。

33、copy_mem（）和 copy_page_tables（）在第一次调用时是如何运行的？

copy_mem()的第一次调用是进程 0 创建进程 1 时，它先提取当前进程（进程 0）的代码段、数据段的段限长，并将当前进程（进程 0）的段限长赋值给子进程（进程 1）的段限长。然后提取当前进程（进程 0）的代码段、数据段的段基址，检查当前进程（进程 0）的段基址、段限长是否有问题。接着设置子进程（进程 1）的 LDT 段描述符中代码段和数据段的基地址为 nr(1)*64MB。最后调用 copy_page_table()函数

copy_page_table()的参数是源地址、目的地址和大小，首先检测源地址和目的地址是

否都是 4MB 的整数倍，如不是则报错，不符合分页要求。然后取源地址和目的地址所对应的页目录项地址，检测如目的地址所对应的页目录表项已被使用则报错，其中源地址不一定是连续使用的，所以有不存在的跳过。接着，取源地址的页表地址，并为目的地址申请一个新页作为子进程的页表，且修改为已使用。然后，判断是否源地址为 0，即父进程是否为进程 0 ，如果是，则复制页表项数为 160，否则为 1k。最后将源页表项复制给目的页表，其中将目的页表项内的页设为“只读”，源页表项内的页地址超过 1M 的部分也设为"只读"（由于是第一次调用，所以父进程是 0，都在 1M 内，所以都不设为“只读”），并在 mem_map 中所对应的项引用计数加 1。1M 内的内核区不参与用户分页管理。

34、缺页中断是如何产生的，页写保护中断是如何产生的，操作系统是如何处理的？

每一个页目录项或页表项的最后 3 位，标志着所管理的页面的属性，分别是 U/S,R/W,P.如果和一个页面建立了映射关系，P 标志就设置为 1，如果没有建立映射关系，则就是 0.进程执行时，线性地址被 MMU 即系，如果解析出某个表项的 P 位为 0，就说明没有对应页面，此时就会产生缺页中断。

当两个进程共享了一个页面，即 R/w 为 1，导致该页面设为“只读”属性，当其中一个进程需要压栈时就会引发页写保护中断。当页写保护中断产生时，系统会为进程申请新页面，并把原页面内容复制到新页面里。

35、为什么要设计缓冲区，有什么好处？（P310）

缓冲区是内存与外设（块设备，如硬盘等）进行数据交互的媒介。内存与外设最大的区别在于：外设（如硬盘）的作用仅仅就是对数据信息以逻辑块的形式进行断电保存，并不参与运算（因为 CPU 无法到硬盘上进行寻址）；而内存除了需要对数据进行保存以外，还要通过与 CPU 和总线的配合，进行数据运算（有代码和数据之分）；缓冲区则介于两者之间，有了缓冲区这个媒介以后，对外设而言，它仅需要考虑与缓冲区进行数据交互是否符合要求，而不需要考虑内存中内核、进程如何使用这些数据；对内存的内核、进程而言，它也仅需要考虑与缓冲区交互的条件是否成熟，而并不需要关心此时外设对缓冲区的交互情况。它们两者的组织、管理和协调将由操作系统统一操作，这样就大大降低了数据处理的维护成本。

缓冲区的好处主要有两点：①形成所有块设备数据的统一集散地，操作系统的设计更方便、更灵活；②对块设备的文件操作运行效率更高。

36、操作系统如何利用 buffer_head 中的 b_data，b_blocknr，b_dev，b_uptodate，b_dirt，b_count，b_lock，b_wait 管理缓冲块的？

buffer_head 负责进程与缓冲块的数据交互，让数据在缓冲区中停留的时间尽可能长。

b_data 是缓冲块的数据内容。

b_dev 和 b_blocknr 两个字段把缓冲块和硬盘数据块的关系绑定，同时根据 b_count 决定是否废除旧缓冲块而新建缓冲块以保证数据在缓冲区停留时间尽量长。

b_dev 为设备标示，b_blocknr 标示 block 块好。b_count 用于记录缓冲块被多少个进程共享了。

b_uptodate 和 b_dirt 用以保证缓冲块和数据块的正确性。b_uptodate 为 1 说明缓冲块的数据就是数据块中最新的，进程可以共享缓冲块中的数据。b_dirt 为 1 时说明缓冲块数据已被进程修改，需要同步到硬盘上。

b_lock 为 1 时说明缓冲块与数据块在同步数据，此时内核会拦截进程对该缓冲块的操作，直到交互结束才置 0。b_wait 用于记录因为 b_lock=1 而挂起等待缓冲块的进程数。

37、操作系统如何利用 b_uptodate 保证缓冲块数据的正确性？new_block (int dev)函数新申请一个缓冲块后，并没有读盘，b_uptodate 却被置 1，是否会引起数据混乱？详细分析理由。

b_uptodate 是缓冲块中针对进程方向的标志位，它的作用是告诉内核，缓冲块的数据是否已是数据块中最新的。当 b_update 置 1 时，就说明缓冲块中的数据是基于硬盘数据块的，内核可以放心地支持进程与缓冲块进行数据交互；如果 b_uptodate 为 0，就提醒内核缓冲块并没有用绑定的数据块中的数据更新，不支持进程共享该缓冲块。

当为文件创建新数据块，新建一个缓冲块时，b_uptodate 被置 1，但并不会引起数据混乱。此时，新建的数据块只可能有两个用途，一个是存储文件内容，一个是存储文件的 i_zone 的间接块管理信息。

如果是存储文件内容，由于新建数据块和新建硬盘数据块，此时都是垃圾数据，都不是硬盘所需要的，无所谓数据是否更新，结果“等效于”更新问题已经解决。

如果是存储文件的间接块管理信息，必须清零，表示没有索引间接数据块，否则垃圾数据会导致索引错误，破坏文件操作的正确性。虽然缓冲块与硬盘数据块的数据不一致，但同样将 b_uptodate 置 1 不会有问题。

综合以上考虑，设计者采用的策略是，只要为新建的数据块新申请了缓冲块，不管这个缓冲块将来用作什么，反正进程现在不需要里面的数据，干脆全部清零。这样不管与之绑定的数据块用来存储什么信息，都无所谓，将该缓冲块的 b_uptodate 字段设置为 1，更新问题“等效于”已解决

额外补充习题（可能作为附加题）

+1. 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。已经有 2000 多个 buffer_head 挂到 hash_table 上时，hash 表（包括所有的 buffer_head）的整体运行状态。

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？

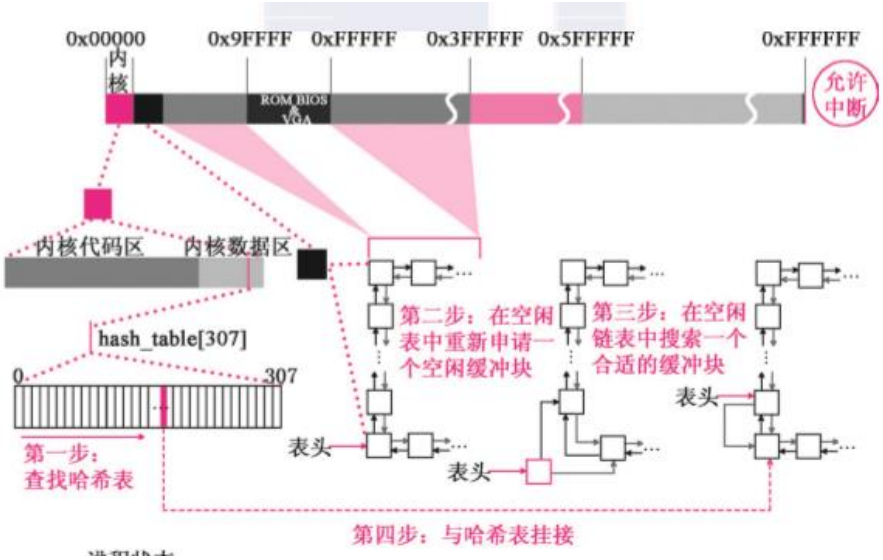
A

getblk(int dev, int block) → get_hash_table(dev,block) -> find_buffer(dev,block) -> hash(dev, block)

哈希策略为：

```
#define _hashfn(dev,block)(((unsigned)(dev block))%NR_HASH)
#define hash(dev,block) hash_table[_hashfn(dev, block)]
```

此时，dev 为 0x300，block 为 0，NR_HASH 为 307，哈希结果为 154，将此块插入哈希表中次位置后



B

//代码路径 ： fs/buffer.c:

…

```
static inline void insert_into_queues(struct buffer_head * bh) {
/*put at end of free list */
        bh->b_next_free= free_list;
        bh->b_prev_free= free_list->b_prev_free;
        free_list->b_prev_free->b_next_free= bh;
        free_list->b_prev_free= bh;
/*put the buffer in new hash-queue if it has a device */
        bh->b_prev= NULL;
        bh->b_next= NULL;
        if (!bh->b_dev)
                return;
        bh->b_next= hash(bh->b_dev,bh->b_blocknr);
        hash(bh->b_dev,bh->b_blocknr)= bh;
        bh->b_next->b_prev= bh
        }
```

C

不会脱钩，会调用 brelse()函数，其中 if(!(buf->b_count--))，计数器减一。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，因此该空闲缓冲块若是能够再次被访问到，对提升性能是有益的。

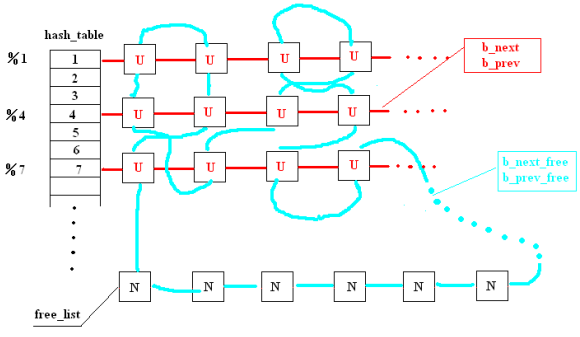
D

进程顺着 freelist 找到没被占用的，未被上锁的干净的缓冲块后，将其引用计数置为 1，然后从哈西队列和空闲块链表中移除该 bh，然后根据此新的设备号和块号重新插入空闲表和哈西队列新位置处，最终返回缓冲头指针。

Bh->b_count=1;

Bh->b_dirt=0;

总的效果图



Bh->b_uptodate=0;
Remove_from_queues(bh);
Bh->b_dev=dev;
Bh->b_blocknr=block;
Insert_into_queues(bh);

+2. 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快、超级快、i 节点位图、逻辑块位图、i 节点、逻辑块？

虚拟盘中没有引导快、超级快、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数据，仅是一段被'\0'填充的内存空间。

（代码路径：kernel/blk_dev/ramdisk.c　　rd_load:）

Rd_start = (char *)mem_start;
Rd_length = length;
Cp = rd_start;
For (i=0; i<length; i++)
　　*cp++='\0\;

+3. 在虚拟盘被设置为根设备之前，操作系统的根设备是软盘，请说明设置软盘为根设备的技术路线。

首先，将软盘的第一个山区设置为可引导扇区:

（代码路径：boot/bootsect.s）　　boot_flag: .word 0xAA55

在主 Makefile 文件中设置 ROOT_DEV=/dev/hd6。并且在 bootsect.s 中的 508 和 509 处设置 ROOT_DEV=0x306；在 tools/build 中根据 Makefile 中的 ROOT_DEV 设置 MAJOR_TOOT 和 MINOR_ROOT，并将其填充在偏移量为 508 和 509 处：

(代码路径：Makefile)　　tools/build　　boot/bootsect　　boot/setup　　tools/system $(ROOT_DEV) > Image

随后被移至 0x90000+508(即 0x901FC)处，最终在 main.c 中设置为 ORIG_ROOT_DEV 并将其赋给 ROOT_DEV 变量：

(代码路径：init/main.c)

62 #define ORIG_ROOT_DEV (*(unsigned short *)0x901FC)
113 ROOT_DEV = ORIG_ROOT_DEV;

+4. 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？

rd_load 函数从软盘读取文件系统并将其复制到虚拟盘中并通过设置 ROOT_DEV 为 0x0101 将根设备从软盘更换为虚拟盘，然后调用 mount_root 函数加载跟文件系统，过程如下：初始化 file_table 和 super_block，初始化 super_block 并读取根 i 节点，然后统计空闲逻辑块数及空闲 i 节点数：

(代码路径：kernel/blk_drv/ramdisk.c:rd_load)　　ROOT_DEV=0x0101;

主设备好是 1，代表内存，即将内存虚拟盘设置为根目录。

+5 在 Linux 操作系统中大量的使用了中断、异常类的处理，为什么，有什么好处？

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响应中断信号而建立的。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“被动模式”代替“主动轮询”模式来处理终端问题是现在操作系统之所以称之为“现代”的一个重要标志。

+6 详细分析一个进程从创建、加载程序、执行、退出的全过程。

1. 创建进程，调用 fork 函数。
   a) 准备阶段，为进程在 task[64]找到空闲位置，即 find_empty_process（）；
   b) 为进程管理结构找到储存空间：task_struct 和内核栈。
   c) 父进程为子进程复制 task_struct 结构
   d) 复制新进程的页表并设置其对应的页目录项
   e) 分段和分页以及文件继承。
   f) 建立新进程与全局描述符表（GDT）的关联
   g) 将新进程设为就绪态
2. 加载进程
   a) 检查参数和外部环境变量和可执行文件
   b) 释放进程的页表
   c) 重新设置进程的程序代码段和数据段
   d) 调整进程的 task_struct
3. 进程运行
   a) 产生缺页中断并由操作系统响应
   b) 为进程申请一个内存页面
   c) 将程序代码加载到新分配的页面中
   d) 将物理内存地址与线性地址空间对应起来
   e) 不断通过缺页中断加载进程的全部内容
   f) 运行时如果进程内存不足继续产生缺页中断，
4. 进程退出
   a) 进程先处理退出事务
   b) 释放进程所占页面
   c) 解除进程与文件有关的内容并给父进程发信号
   d) 进程退出后执行进程调度

+7 详细分析多个进程（无父子关系）共享一个可执行程序的完整过程。

假设有三个进程 A、B、C，进程 A 先执行，之后是 B 最后是 C，它们没有父子关系。A 进程启动后会调用 open 函数打开该可执行文件，然后调用 sys_read()函数读取文件内容,该函数最终会调用 bread 函数，该函数会分配缓冲块，进行设备到缓冲块的数据交换，因为此时为设备读入，时间较长，所以会给该缓冲块加锁，调用 sleep_on 函数，A 进程被挂起，调用 schedule()函数 B 进程开始执行。

B 进程也首先执行 open（）函数，虽然 A 和 B 打开的是相同的文件，但是彼此操作没有关系，所以 B 继承需要另外一套文件管理信息，通过 open_namei()函数。B 进程调用 read 函数，同样会调用 bread（），由于此时内核检测到 B 进程需要读的数据已经进入缓冲区中，则直接返回，但是由于此时设备读没有完成，缓冲块以备加锁，所以 B 将因为等待而被系

统挂起，之后调用 schedule()函数。
C 进程开始执行，但是同 B 一样，被系统挂起，调用 schedule()函数，假设此时无其它进程，则系统 0 进程开始执行。
假设此时读操作完成，外设产生中断，中断服务程序开始工作。它给读取的文件缓冲区解锁并调用 wake_up()函数，传递的参数是&bh->b_wait,该函数首先将 C 唤醒，此后中断服务程序结束，开始进程调度，此时 C 就绪，C 程序开始执行，首先将 B 进程设为就绪态。C 执行结束或者 C 的时间片削减为 0 时，切换到 B 进程执行。进程 B 也在 sleep_on()函数中，调用 schedule 函数进程进程切换，B 最终回到 sleep_on 函数，进程 B 开始执行，首先将进程 A 设为就绪态，同理当 B 执行完或者时间片削减为 0 时，切换到 A 执行，此时 A 的内核栈中 tmp 对应 NULL，不会再唤醒进程了。

+8 为什么 get_free_page（）将新分配的页面清 0？（P265）
因为无法预知这页内存的用途，如果用作页表，不清零就有垃圾值，就是隐患。

+9.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？
剩余的内容： 0x5400~0x54b7 处包含了 after_page_tables、 ignore_int 中断服务程序和 setup_paging 设 置分页的代码。
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: 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:为设置分页机制的代码，它在分页完成前不能被覆盖

+10.进程 0 的 task_struct 在哪？具体内容是什么？给出代码证据。
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：（课本 P68）