第一部分： 内核代码前

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

2、setup程序里的cli是为了什么？

3、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？

4、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？

5、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？

6、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。

7、保护模式的“保护”体现在哪里？

8、特权级的目的和意义是什么？为什么特权级是基于段的？

9、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？

10、用户进程自己设计一套LDT表，并与GDT挂接，是否可行，为什么？

11、保护模式、分页下，线性地址到物理地址的转化过程是什么？

12、为什么开始启动计算机的时候，执行的是BIOS代码而不是操作系统自身的代码？

13、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没有直接把所有需要加载的扇区都加载？

14、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？

第二部分： 内核代码（进程创建）

1、进程0的task\_struct在哪？具体内容是什么？

2、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。

3、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。

4、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，3特权级的进程代码并不能使用cli指令，这是为什么？请解释并给出代码证据。

5、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)  
读懂代码。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。

6、进程0 fork进程1之前，为什么先调用move\_to\_user\_mode()？用的是什么方法？解释其中的道理。

7、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。

8、分析get\_free\_page()函数的代码，叙述在主内存中获取一个空闲页的技术路线。

9、分析copy\_page\_tables（）函数的代码，叙述父进程如何为子进程复制页表。

10、进程0创建进程1时，为进程1建立了task\_struct及内核栈，第一个页表，分别位于物理内存16MB顶端倒数第一页、第二页。请问，这两个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（可以图示）并给出代码证据。

11、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。

12、#define switch\_to(n) {\  
struct {long a,b;} \_\_tmp; \  
\_\_asm\_\_("cmpl %%ecx,\_current\n\t" \  
    "je 1f\n\t" \  
    "movw %%dx,%1\n\t" \  
    "xchgl %%ecx,\_current\n\t" \  
    "ljmp %0\n\t" \  
    "cmpl %%ecx,\_last\_task\_used\_math\n\t" \  
    "jne 1f\n\t" \  
    "clts\n" \  
    "1:" \  
    ::"m" (\*&\_\_tmp.a),"m" (\*&\_\_tmp.b), \  
    "d" (\_TSS(n)),"c" ((long) task[n])); \  
}  
代码中的"ljmp %0\n\t" 很奇怪，按理说jmp指令跳转到得位置应该是一条指令的地址，可是这行代码却跳到了"m" (\*&\_\_tmp.a)，这明明是一个数据的地址，更奇怪的，这行代码竟然能正确执行。请论述其中的道理。

13、进程0开始创建进程1，调用fork（），跟踪代码时我们发现，fork代码执行了两次，第一次，执行fork代码后，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到向转向fork的goto语句，也没有看到循环语句，是什么原因导致fork反复执行？请说明理由（可以图示），并给出代码证据。

14、打开保护模式、分页后，线性地址到物理地址是如何转换的？

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

16、wait\_on\_buffer函数中为什么不用if（）而是用while（）？

17、add\_reques（）函数中有下列代码  
    if (!(tmp = dev->current\_request)) {  
        dev->current\_request = req;  
        sti();  
        (dev->request\_fn)();  
        return;  
    }  
其中的  
    if (!(tmp = dev->current\_request)) {  
        dev->current\_request = req;  
是什么意思？

18、do\_hd\_request()函数中dev的含义始终一样吗？

19、read\_intr（）函数中，下列代码是什么意思？为什么这样做？  
    if (--CURRENT->nr\_sectors) {  
        do\_hd = &read\_intr;  
        return;  
    }

20、bread（）函数代码中为什么要做第二次if (bh->b\_uptodate)判断？  
    if (bh->b\_uptodate)  
        return bh;  
    ll\_rw\_block(READ,bh);  
    wait\_on\_buffer(bh);  
    if (bh->b\_uptodate)  
        return bh;

21、getblk（）函数中，两次调用wait\_on\_buffer（）函数，两次的意思一样吗？

22、getblk（）函数中  
    do {  
        if (tmp->b\_count)  
            continue;  
        if (!bh || BADNESS(tmp)<BADNESS(bh)) {  
            bh = tmp;  
            if (!BADNESS(tmp))  
                break;  
        }  
/\* and repeat until we find something good \*/  
    } while ((tmp = tmp->b\_next\_free) != free\_list);  
说明什么情况下执行continue、break。

23、make\_request（）函数      
    if (req < request) {  
        if (rw\_ahead) {  
            unlock\_buffer(bh);  
            return;  
        }  
        sleep\_on(&wait\_for\_request);  
        goto repeat;  
其中的sleep\_on(&wait\_for\_request)是谁在等？等什么？