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（直接从物理地址分配）？说明理由并给出代码证据。

分析：两次都是通过调用get_free_page()在物理内存里申请一个物理页，由于在head.s中决定内核的物理地址和线性地址是一一对应的。因此这两个页都在内核的线性地址空间内。［理解：页占用了谁的线性地址空间其实就是表示该页是归谁管，此时这两个页虽然是分配给进程1的，但是管理它的确实内核，就比如身份证和警察局的关系，进程0也是如此，其task_struct和页表均归属内核的线性地址空间；此外进程0的结构在内核数据区部分］

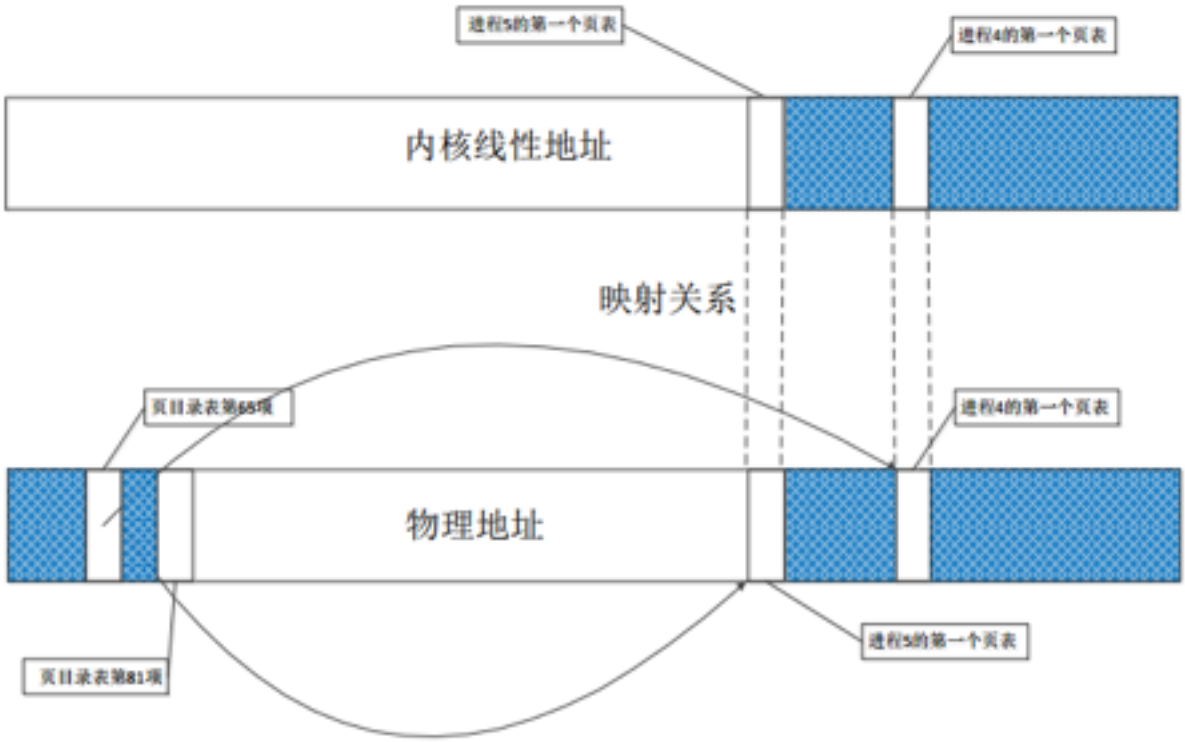
setup_paging: （P39）

```
        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              /*  --------- " " --------- */
        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
```

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。

分析：1)内核的线性地址空间。

2)进程 4 中的两个页面挂载在页目录表第 65项指向的页表的最后两项,进程 5 中的两个页面挂载在页目录表第 81项指向的页表的最后两项。



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。

分析：第一次fork执行的时候，eax值为创建的进程号1（P88中find_empty_process函数返回以及P102说明），fork()汗水为1，判断为假，故会跳过init执行for循环；在这个阶段，0x08中断对应进行了CPU的ss、esp、eflags、cs、eip寄存器的压栈动作，此处eip保存的就是进程0的fork代码中if位置；pause阶段会执行schedule函数切换到进程1，并设置进程0为可中断等待状态（P104往后）；而在sche中switch_to函数的ljmp通过任务们机制，保存了当前进程－进程0－的tss状态，同时将进程1的tss恢复给CPU继续执行，此时实现了从0特权集到3的切换；（P109往前）

另一方面第一次fork时中断后续的函数对进程1做了相关操作，其中重要的一项是在copy_process函数中设置了进程1的tss.eip指向当前进程0指向的位置，即if的判断位置，以及tss.eax等于0。前者保证进程1在ljmp切换继续执行的时候直接执行到if(__res>=0)处（此即第二次fork），而eax为0则表示此处fork函数返回为0，即（!fork()）为真，故进入init函数执行。此时在copy_mem函数中进行的进程0到进程1的task_struct的复制160个页（P93-98），即对进程0和进程1进行了数据共享（包括代码）

此时，进程0的第一次fork中断状态，进程1切换执行，进程0并没有中断返回还

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。

分析：这几个压栈动作是中断int 0x80 由硬件强制执行的，故么有压栈代码

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

分析：定义成inline函数一方面inline函数仍旧保留了真正函数的特性，在预编译的时候会进行相关检查；同时定义成inline函数有利于后面的函数名扩展；另一方面，inline函数在实际调用过程时已经做了代码展开工作，也就没有对应的call和ret动作，那么就不需要保存返回时函数地址，即EIP的值；否则如果没有inline关键字，那么此时调用syscall0函数会对应在进程0的栈中压入函数返回时的地址，即修改了EIP值，在后续的执行copy_process所进行的给进程1的tss赋值操作这里的EIP已经给修改成了syscall0的返回地址，而非原来的EIP值，那么第二次fork返回时便会出错

6.根据代码详细说明copy_process函数的所有参数是如何形成的?

分析：ss、esp、eflags、cs、eip是0x80中断时硬件强制压栈的；中断int 0x80-系统调用总入口_system_call，压入了ds、es、fs、edx、ecx、ebx（P85）；由于在_system_call中_sys_call_table也会有一个压栈动作保护现场，由于copy_process参数和栈中内容一一对应，故此处需要补充一个none参数；查找第一个空闲进程时压入了余下的几个即gs、esi、edi、ebp、eax，此处eax为find_empty_process返回的进程号，此处为进程1，即返回为1，对应于参数nr（P86）

7.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。

分析：参考题4以及P104

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。

分析： P135和P138

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、第一个页表的前7个页表项指向什么位置？给出 代码证据。

分析：每一个页大小为4KB，页目录表为1024项，每一项对应一个页表，每个页表共1024项，每一个项对应1个页，即总共线性地址空间可占有4G；挂接关系如图P39 1-42；

(特别注意页表 1 第一项指向页目录表) Head.s 中:
setup_paging:
movl $1024*5,%ecx xorl %eax,%eax xorl %edi,%edi cld;rep;stosl
movl $pg0+7,pg_dir

/* 5 pages - pg_dir+4 page tables */

/* pg_dir is at 0x000 */

/* set present bit/user r/w */ /* --------- " " --------- */ /* --------- " " --------- */ /* --------- " " --------- */

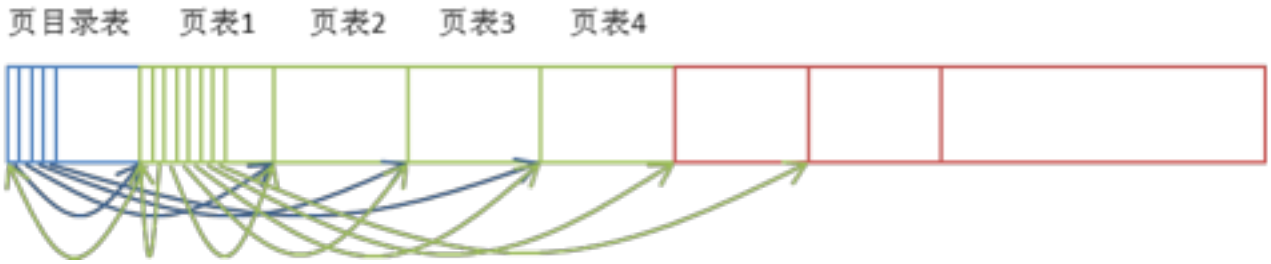
movl $pg1+7,pg_dir+4
movl $pg2+7,pg_dir+8
movl $pg3+7,pg_dir+12
_pg_dir 用于表示内核分页机制完成后的内核起始位置,也就是物理内存的起始位

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



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。

分析：（P51-56）图2-8和图2－9，以及_set_gat汇编代码

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?

分析：因为在计算机启动最开始的时候，内存RAM中没有任何的代码；此时操作系统代码在软盘中，而cpu的逻辑电路被设计为只能运行内存中的程序，没有能力直接从软盘运行操作系统

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?

分析：这是因为BIOS默认在计算机启动的时候通过BIOS中断只从启动扇区(对Linux-0.11而言指软盘的第一个扇区)加载代码。而后续代码的加载则是通过系统中断完成的。因为如果要使用BIOS进行加载，而且加载完成之后再执行，则需要很长的时间；因此Linux采用的是边执行边加载的方法。

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?

分析：因为BIOS首先会把中断向量表加载到0x00000-0x003ff的1KB的内存空间，在加载bootsect时约定加载到0x07c00处。挪到0x90000处是操作系统自身的代码完成的，是为加载后续代码做准备；而加载到0x07c00是BIOS规范，操作系统也没办法

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

分析：bootsect->setup->head分别通过jmpi 0，SETUPSEG（P15）和jmp 0，0x8（P25）

15.setup程序里的cli是为了什么?

分析：向32位模式转变时，使用cli时进行关中断（P17点评）

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?

分析：因为此时已经切换到32位保护模式下来了，故需要遵循此时的寻址规则，段基址＋段偏移的方式，此时8应该理解成1000，第一位表示gdt中的第2项，第二位表示gdt，后两位表示特权级0，即此时跳转到gdt第二项数据所指定的位置开始执行，根据P26的图1-23即可寻址到head.s处

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?

分析：打开A20，也就是把原来的20根地址线变成了32根地址线，意味着能够进行32寻址，CPU的最大寻址空间变成了4GB；PE时CR0控制寄存器的第0位，设置为0即变成了保护模式。这是有必要的，保护模式的一个重要特征就是根据GDT来决定后续执行哪里的程序，A20只是个标志而已，未真正变成32位寻址模式

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?

分析：C语言编写的程序都是用户应用程序，这类代码都必须在操作系统的平台上执行，也就是说，要由操作系统为应用程序创建进程，并把程序的可执行代码从硬盘上加载到内存。而操作系统的加载则是由BIOS完成的，bootsect.s和setup.s涌来加载操作系统内核代码，而另一方面，此时BIOS所形成的是16的实模式，linux是一个32位的实时多任务的现代操作系统，main函数肯定需要执行32位的代码，故head.s的执行即来弥补16位到32位的空缺

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。

分析：因为在由head程序向main函数跳转时，是不需要main函数返回的；这是由于main函数已经是最底层的函数了，没有更底层的支撑函数支持其返回。所以Linux采用ret指令，模拟函数返回，跳转到main函数去执行。（P42）

20.保护模式的“保护”体现在哪里?

分析：

1) 段。

. 2) 利用分段机制和分页机制,每个程序放在不同的虚拟地址(逻辑地址)空间中,每 个程序有自己的虚拟地址和物理地址的映射表,防止一个程序访问另一个程序或操 作系统的内存区域。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。

3) 限制任务访问权限,保护操作系统内存段和处理器特殊系统寄存器不被应用程序访 问。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。(为什么基于段: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)

4) 有,进程的页表只有内核可见,因此其它进程也无法访问本进程的线性空间。

21.特权级的目的和意义是什么？为什么特权级是基于段的?

分析：特权级是操作系统为了更好的管理内存空间而设的，提高了系统的安全性。通过段，系统划分了内核代码段、内核数据段、用户代码段和用户数据段等不同的数据段，有些段是系统专享的，有些是和用户程序共享的，因此就有特权级的概念

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?

分析：P33

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?

分析：通过运行Linux-0.11代码，可以看到剩余的代码是标号after_page_tables之后head程序中的代码。因为head程序共占用25KB+184B的内存空间，系统在建立好分页机制和GDT、IDT之后，在内存空间0x05400-0x54b8处留有184B的空间未使用，因此产生了剩余。（P40）

24.进程0的task_struct在哪？具体内容是什么？给出代码证据。

分析：在内核数据段部分；P66代码 + P68 INIT_TASK

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。

分析：dpl为3是表示系统调用可以由3特权级（即用户进程进行调用）（P55）系统调用是设计给用户态进程调用,用户态的进程的特权级为 3,要进行系统调 用时如果 DPL 为 0,那进程无法进行系统调用。中断和陷阱的服务程序中有许多内核态才可执行的特权指令,因此要求进程处于内核态时才可调用;这样设计能防止内核被破坏。［dpl表示的是特权级，0和3分别表示0特权级和3特权级。异常处理是由内核来完成的，Linux出于对内核的保护，不允许用户进程直接访问内核。但是有些情况下，用户进程又需要内核代码的支持，因此就需要系统调用，它是用户进程与内核打交道的接口，是由用户进程直接调用的。因此其在3特权级下。］

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 。

分析：从0特全级切换到3特权级：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。

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。

分析：task_struct和页表（P89+P98）

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。

分析：P68－－进程0代码INIT_TASK中的eflags的值此时已经决定好了；［CPL、IOPL 和控制寄存器 CR4 中的 VME 标志决定着 IF 标志是否可由 CLI、STI、 POPF、POPFD 和 IRET 指令修改。 IF 标志分别用 STI 和 CLI 指令设置或清除。只有当 CPL 小于或等于 IOPL 时才可以执行这两个指令。如果在 CPL 大于 IOPL 的情况下执行, 将会产生一个一般保护异常(#GP)。3 特权级的进程 CPL 为 3,而 kernel TSS 中 elags 中 IOPL 为 0,IOPL 被设为 0,所以 CLI 只能在 0 特权级使用。(代码见书 68 页 INIT_TASK 中 TSS 对 eflags 的赋值)。 ］

30.根据代码详细分析操作系统是如何获得一个空闲页的。

分析：全局的task数组保存了所有进程的task_struct结构，遍历并比较last_pid获得第一个空闲的任务号，参考函数find_empty_process代码（P77-P88）