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。**

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

bootsect首先利用int 0x13中断分别加载setup程序及system模块，待bootsect程序的任务完成之后，执行代码 jmpi 0,SETUPSEG。由于 bootsect 将 setup段加载到了 SETUPSEG:0 （0x90200）的地方,在实模式下，CS:IP 指向 setup程序的第一条指令，此时 setup 开始执行。

② setup 跳转到 head 程序：jmpi 0,8

执行 setup 后，内核被移到了 0x00000 处，CPU 变为保护模式，执行 jmpi 0,8并加载了中断描述符表和全局描述符表。该指令执行后跳转到以 GDT 第 2 项 中的 base\_addr 为基地址，以 0 为偏移量的位置，其中 base\_addr 为 0。由于head 放置在内核的头部，因此程序跳转到 head 中执行。

1. **setup程序里的cli是为了什么？**

答：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。

**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？**

此时为 32 位保护模式，“0”表示段内偏移，“8”表示段选择符。转化为二进制：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 表示根据 GDT中的第 2 项来确定代码段的段基址和段限长等信息。可以得到代码是从 head 的开始位置，段基址 0x00000000、偏移为 0 处开始执行的。

1. **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？**

有必要。

A20 是 CPU 的第 21 位地址线，A20 未打开的时候，实模式下的最大寻址为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。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，而打开 PE 是使能保护模式。打开 A20 是打开 PE 的必要条件；而打开A20 不一定非得打开 PE。打开 PE 是说明系统处于保护模式下，如果不打开 A20的话，A20 会被强制置 0，则保护模式下访问的内存是不连续的，如0~1M,2~3M,4~5M 等，若要真正在保护模式下工作，必须打开 A20，实现 32位寻址。

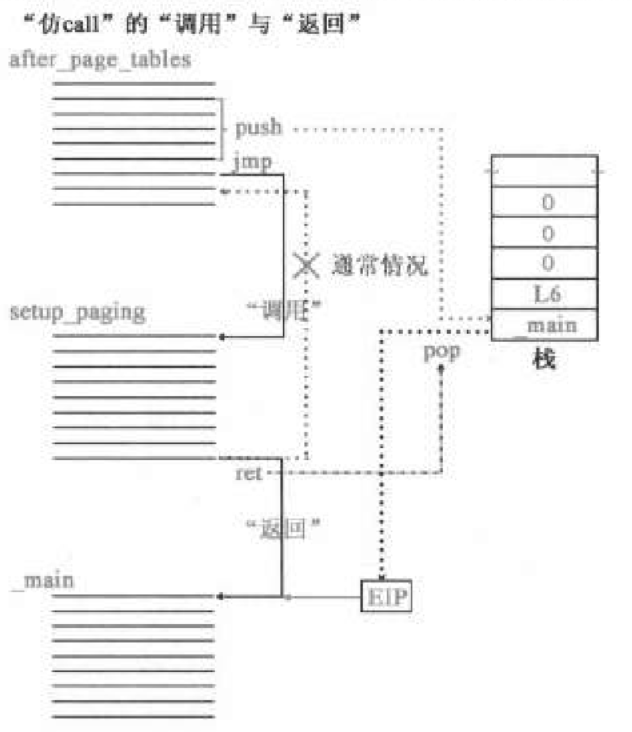
**5、Linux是用C语言写的，为什么没有从main还是开始，而是先运行3个汇编程序，道理何在？**

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 32 位的保护模式状态下时，调用 main 的条件才算准备完毕。

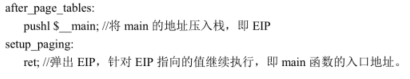
**6、为什么不用call，而是用ret“调用”main函数？画出调用路线图，给出代码证据。**

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，当执行到被调函数的 ret 指令时，自动出栈给 EIP 并还原现场，继续执行 CALL 的下一行指令。在由 head 程序向 main 函数跳转时，不需要 main 函数返回；且因为main 函数是最底层的函数，无更底层的函数进行返回。因此要达到既调用 main又不需返回，选择 ret。

调用路线图：见 P42 图 1-46。仿 call 示意图 下面部分



代码证据：



**7、保护模式的“保护”体现在哪里？**

保护是指操作系统的安全，不受到恶意攻击。保护进程地址空间。

“ 保护”体现在打开保护模式后，CPU 的寻址模式发生了变化，基于 GDT 去获取代码或数据段的基址，相当于增加了一个段位寄存器。防止了对代码或数据段的覆盖以及代码段自身的访问超限。对描述符所描述的对象进行保护：（1）在 GDT、LDT 及 IDT中，均有对应界限、特权级等；（2）在不同特权级间访问时，系统会对 CPL、 RPL、DPL、 IOPL 等进行检验，同时限制某些特殊指令如 lgdt, lidt,cli 等的使用；（3）分页机制中 PDE 和 PTE 中的 R/W 和 U/S 等提供了页级保护，分页机制通过将线性地址与物理地址的映射，提供了对物理地址的保护。

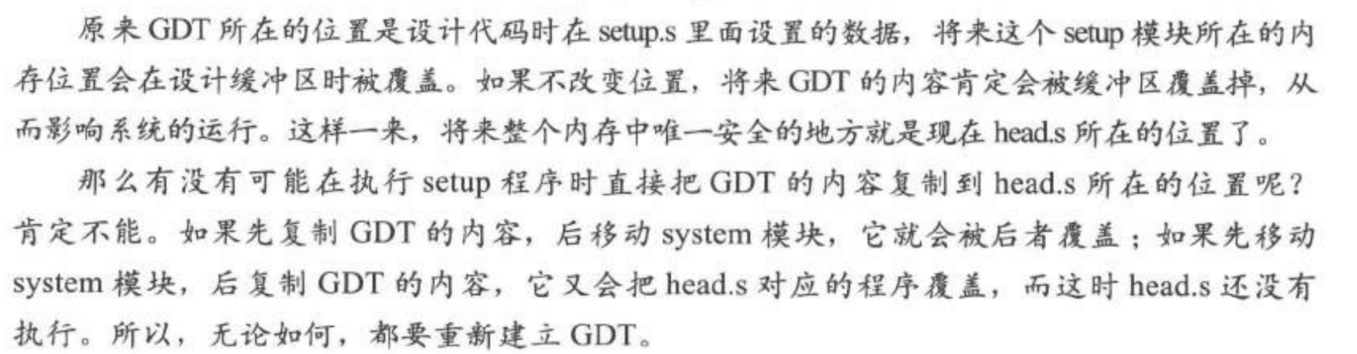
**8、特权级的目的和意义是什么？为什么特权级是基于段的？**

特权级机制目的是为了进行合理的管理资源，保护高特权级的段。

意义是进行了对系统的保护，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部分关键性指令，通过特权级的设置禁止用户进程使用cli、 sti 等指令。将内核设计成最高特权级，用户进程成为最低特权级。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，特权级的引入对操作系统内核进行保护。

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。

**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？**

不可行

GDT 和 LDT 放在内核数据区，属于 0 特权级，3 特权级的用户进程无权访问修改。此外，如果用户进程可以自己设计 LDT 的话，表明用户进程可以访问其他进程的 LDT，则会削弱进程之间的保护边界，容易引发问题。

补充：

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。但是真正其作用的 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。而对 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。LDT 表只是一段内存区域，我们可以构造出用户空间的 LDT。而且 Ring0 代码可以访问 Ring3 数据。但是这并代表我们的用户空间 LDT 可以被挂载到 GDT上。考察挂接函数 set\_ldt\_desc：1）它是 Ring0 代码，用户空间程序不能直接调用；2）该函数第一个参数是 gdt 地址，这是 Ring3 代码无权访问的，又因为gdt 很可能不在用户进程地址空间，就算有权限也是没有办法寻址的。3）加载ldt 所用到的特权指令 lldt 也不是 Ring3 代码可以任意使用的。

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？**

计算机加电启动的时候，操作系统并没有在内存中，CPU 也不能从外设运行操作系统，所以必须将操作系统加载到内存中，该过程的最开始部分是由 BIOS 的中断完成的。在加电后，BIOS 需要完成一些检测工作，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至 0x7C00 处，然后将跳转至 0x7C00 运行操作系统自身的代码。所以计算机启动最开始运行的是 BIOS 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？**

BIOS 和操作系统的开发通常是不同的团队，按固定的规则约定，可以进行灵活的各自设计相应的部分。BIOS 接到启动操作系统命令后，只从启动扇区将代码加载至 0x7c00(BOOTSEG)位置，而后续扇区由 bootsect 代码加载，这些代码由编写系统的用户负责，与之前 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。

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。

**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？**

加载0x07c00是BIOS提前约定设置的，不能加载到0x00000是因为从0x00000

开始到 0x003ff 这 1KB 内存空间都是 BIOS 首先约定进行加载中断向量表的地

方，不能进行覆盖。

而后挪到 0x90000 处是操作系统开始根据自己的需要安排内存了，具体原因如

下：

1. 内核会使用启动扇区中的一些数据，如第 508、509 字节处的 ROOT\_DEV；
2. 依据系统对内存的规划，内核占用 0x00000 开始的空间，因此 0x07c00 可

能会被覆盖。

第二部分： 内核代码（进程创建）

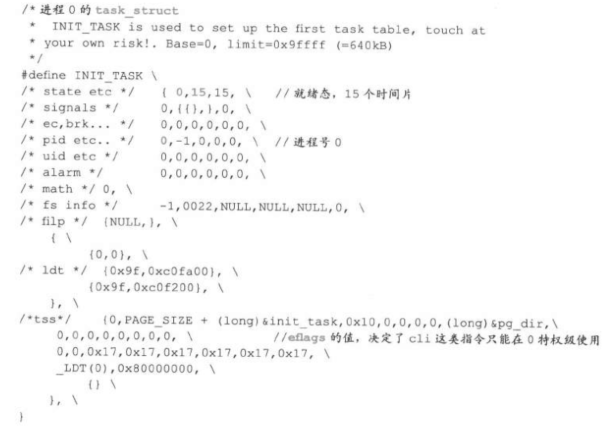
1. **进程0的task\_struct在哪？具体内容是什么？**

进程 0 的 task\_struct 位于内核数据区，因为在进程 0 未激活之前，使用的是 boot阶段的 user\_stack，因此存储在 user\_stack 中。

具体内容如下：

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。

代码如下：(若未要求没时间可不写)

1. **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 在 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。

挂接关系图

代码证据

P39 最下面

1. **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。**

（先画图见 P54 图 2-9 然后解释）以 set\_trap\_gate(0,&divide\_error)为例，其中，n 是 0，gate\_addr 是&idt[0]，也就是 idt 的第一项中断描述符的地址；type是 15，dpl（描述符特权级）是 0；addr 是中断服务程序 divide\_error(void)的入口地址。

1. **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。**

根据 Intel Manual，cli 和 sti 指令与 CPL 和 EFLAGS[IOPL]有关。通过 IOPL 来加以保护指令 in,ins,out,outs,cli,sti 等 I/O 敏感指令，只有 CPL(当前特权级)<=IOPL 才能执行，低特权级访问这些指令将会产生一个一般性保护异常。IOPL 位于 EFLAGS 的 12-13 位，仅可通过 iret 来改变，INIT\_TASK 中 IOPL 为0，在 move\_to\_user\_mode 中直接执行“pushfl \n\t”指令，继承了内核的EFLAGS。IOPL 的指令仍然为 0 没有改变，所以用户进程无法调用 cli 指令。因此，通过设置 IOPL， 3 特权级的进程代码不能使用 cli 等 I/O 敏感指令。

具体代码：move\_to\_user\_mode()分 此处一共两部分代码第一部分 P79

#define move\_to\_user\_mode() \

\_\_asm\_\_(“movl %%esp, %%eax\n\t” \

……

“pushfl\n\t” \ // ELAGS 进栈

……

”)

第二部分代码见 P 68 INIT\_TASK

**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。**

当用户程序产生系统调用软中断后， 系统都通过 system\_call 总入口找到具体的系统调用函数。 set\_system\_gate 设置系统调用，须将 DPL 设置为 3，允许在用户特权级（3）的进程调用，否则会引发 General Protection 异常。

set\_trap\_gate 及 set\_intr\_gate 设置陷阱和中断为内核使用，需禁止用户进程调用，所以 DPL 为 0。

1. **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。**

Linux 操作系统规定，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。需要将进程 0 通过调用 move\_to\_user\_mode()从内核态转换为用户态。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。设计者通过代码模拟 int（中断） 压栈， 当执行 iret 指令时， CPU 将SS,ESP,EFLAGS,CS,EIP 5 个寄存器的值按序恢复给 CPU， CPU 之后翻转到 3 特权级去执行代码。

1. **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。**

在 fork()中， 当执行“int $0x80” 时产生一个软中断， 使 CPU 硬件自动将 SS、 ESP、EFLAGS、 CS、 EIP 这 5 个寄存器的数值按这个顺序压入进程 0 的内核栈。 硬件压栈可确保 eip 的值指向正确的指令， 使中断返回后程序能继续执行。因为通过栈进行函数传递参数，所以恰可做为 copy\_process 的最后五项参数。

1. **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。**

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。P89 代码考试不用 代码考试不用看

过程：

① 将 EAX 设置为 0,EDI 设置指向 mem\_map 的最后一项（mem\_map+PAGING\_PAGES-1），std 设置扫描是从高地址向低地址。从mem\_map 的最后一项反向扫描，找出引用次数为 0(AL)的页，如果没有则退出；如果找到，则将找到的页设引用数为 1；

② ECX 左移 12 位得到页的相对地址，加 LOW\_MEM 得到物理地址，将此页最后一个字节的地址赋值给 EDI（LOW\_MEM+4092）；

1. stosl 将 EAX 的值设置到 ES:EDI 所指内存，即反向清零 1024\*32bit，将此页清空；

④ 将页的地址（存放在 EAX）返回。

1. 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。

P97 一段话解释

1. **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。**

答：均占用内核的线性地址空间， 原因如下：

通过逆向扫描页表位图，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

unsigned long get\_free\_page(void)

{register unsigned long \_\_res asm("ax");

\_\_asm\_\_("std ; repne ; scasb\n\t"

"jne 1f\n\t"

"movb $1,1(%%edi)\n\t"

"sall $12,%%ecx\n\t"

"addl %2,%%ecx\n\t"

"movl %%ecx,%%edx\n\t"

"movl $1024,%%ecx\n\t"

"leal 4092(%%edx),%%edi\n\t"

"rep ; stosl\n\t"

" movl %%edx,%%eax\n"

"1: cld"

:"=a" (\_\_res)

:"0" (0),"i" (LOW\_MEM),"c" (PAGING\_PAGES),

"D" (mem\_map+PAGING\_PAGES-1)

);

return \_\_res;

}

进程 0 和进程 1 的 LDT 的 LIMIT 属性将进程 0 和进程 1 的地址空间限定0~640KB， 所以进程 0、 进程 1 均无法访问到这两个页面， 故两页面占用内核的线性地址空间。进程 0 的局部描述符如下

include/linux/sched.h: INIT\_TASK

/\* ldt \*/ {0x9f,0xc0fa00}, \

{0x9f,0xc0f200}, \

内核线性地址等于物理地址(0x00000~0xfffff)， 挂接操作的代码如下

(head.s/setup\_paging)：

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

1. **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。**

内核的线性地址空间

注：65 和 81 应该改成 64 和 80

**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。**

其中 a 对应 EIP，b 对应 CS，ljmp 此时通过 CPU 中的电路进行硬件切换，进程由当前进程切换到进程 n。CPU 将当前寄存器的值保存到当前进程的 TSS中，将进程 n 的 TSS 数据及 LDT 的代码段和数据段描述符恢复给 CPU 的各个寄存器，实现任务切换。

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。**

主要涉及的代码位置如下：

Init/main.c 代码中 P103 —— if 判断

Include/unistd.h 中 P102 —— fork 函数代码

进程 1 TSS 赋值，特别是 eip，eax 赋值

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;

原因

fork 为 inline 函数，其中调用了 sys\_call0，产生 0x80 中断，将 ss, esp,eflags, cs, eip 压栈，其中 eip 为 int 0x80 的下一句的地址。在 copy\_process中，内核将进程 0 的 tss 复制得到进程 1 的 tss，并将进程 1 的 tss.eax 设为 0，而进程 0 中的 eax 为 1。在进程调度时 tss 中的值被恢复至相应寄存器中，包括 eip，eax 等。所以中断返回后，进程 0 和进程 1 均会从 int 0x80的下一句开始执行，即 fork 执行了两次。由于 eax 代表返回值，所以进程 0 和进程 1 会得到不同的返回值，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，并进行进程调度。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。

**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？**

保护模式下，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 10 位表示页目录表中的页目录项，由此得到所在的页表地址。中间 10 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。示意图（P97 图 3-9 线性地址到物理地址映射过程示意图）

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（）？**

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。

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？**

122 页 不一样。

答： 不是一样的。 dev/=5 之前表示当前硬盘的逻辑盘号。 这行代码之后表示的实际的物理设备号

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;  
    }

答案：参照 答案：参照 P131

**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;**

第一次从高速缓冲区中取出指定和设备和块号相符的缓冲块， 判断缓冲块数据是否有效， 有效则返回此块， 正当用。 如果该缓冲块数据无效（更新标志未置位） ， 则发出读设备数据块请求。

第二次，等指定数据块被读入，并且缓冲区解锁，睡眠醒来之后，要重新判断缓冲块是否有效，如果缓冲区中数据有效，则返回缓冲区头指针退出。否则释放该缓冲区返回 NULL,退出。在等待过程中，数据可能已经发生了改变，所以要第

二次判断。

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

**代码在书上 代码在书上 113 和 和 114**

答： 一样。 都是等待缓冲块解锁。第一次调用是在， 已经找到一个比较合适的空闲缓冲块， 但是此块可能是加锁的， 于是等待该缓冲块解锁。

第二次调用， 是找到一个缓冲块， 但是此块被修改过， 即是脏的， 还有其他进程在写或此块等待把数据同步到硬盘上， 写完要加锁， 所以此处的调用仍然是等待缓冲块解锁。

**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。**

（P114 代码）

Continue ：if (tmp->b\_count)在判断缓冲块的引用计数，如果引用计数不为 0，那么继续判断空闲队列中下一个缓冲块（即 continue），直到遍历完。Break ：如果有引用计数为 0 的块，那么判断空闲队列中那些引用计数为 0 的块的 badness，找到一个最小的，如果在寻找的过程中出现 badness 为 0 的块，那么就跳出循环（即 break）。

如果利用函数 get\_hash\_table 找到了能对应上设备号和块号的缓冲块，那么直接返回。

如果找不到，那么就分为三种情况：

1.所有的缓冲块 b\_count=0，缓冲块是新的。

2.虽然有 b\_count=0，但是有数据脏了，未同步或者数据脏了正在同步加和既不

脏又不加锁三种情况；

3.所有的缓冲块都被引用，此时 b\_count 非 0，即为所有的缓冲块都被占用了。综合以上三点可知，如果缓冲块的 b\_count 非 0，则 continue 继续查找，知道找到 b\_count=0 的缓冲块；如果获取空闲的缓冲块，而且既不加锁又不脏，此时 break，停止查找。

**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？**

这行代码是当前进程在等（如：进程 1），在等空闲请求项。

make\_request()函数创建请求项并插入请求队列，执行 if 的内容说明没有找到空

请求项：如果是超前的读写请求，因为是特殊情况则放弃请求直接释放缓冲区，

否则是一般的读写操作，此时等待直到有空闲请求项，然后从 repeat 开始重新

查看是否有空闲的请求项。

**补充内容**

**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。

答案 2：inline 一般是用于定义内联函数，内联函数结合了函数以及宏的优点，

在定义时和函数一样，编译器会对其参数进行检查；在使用时和宏类似，内联函

数的代码会被直接嵌入在它被调用的地方，这样省去了函数调用时的一些额外开

销，比如保存和恢复函数返回地址等，可以加快速度。

**根据代码详细说明 copy\_process 函数的所有参数是如何形成的？ 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。

额外注释：

一般在应用程序中，一个函数的参数是由函数定义的，而在操作系统底层中，函

数参数可以由函数定义以外的程序通过压栈的方式“做”出来。copy\_process 函数

的所有参数正是通过压栈形成的。代码见 P83 页、P85 页、P86 页。

**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？ 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。

**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：

第一次调用 get\_free\_page 函数申请的空闲页面用于进程 1 的 task\_struct 及内

核栈。首先将申请到的页面清 0，然后复制进程 0 的 task\_struct，再针对进程 1

作个性化设置，其中 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。代码见 P90 及 P92。

kenel/fork.c:copy\_process

p = (struct task\_struct \*)get\_free\_page();

\*p = \*current

p->tss.esp0 = PAGE\_SIZE + (long)p;

第二次调用 get\_free\_page 函数申请的空闲页面用于进程 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），同时修改了页表项的属性为只读。代码见 P98。

mm/memory.c: copy\_page\_table

if(!(to\_page\_table = (unsigned long \*)get\_free\_page()))

return -1;

\*to\_dir = ((unsigned long)to\_page\_table) | 7;

**为什么 get\_free\_page （）将新分配的页面清 0 ？P265**

答：

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。

答 2：Linux 在回收页面时并没有将页面清 0，只是将 mem\_map 中与该页对应

的位置 0。在使用 get\_free\_page 申请页时，也是遍历 mem\_map 寻找对应位为

0 的页，但是该页可能存在垃圾数据，如果不清 0 的话，若将该页用做页表，则

可能导致错误的映射，引发错误，所以要将新分配的页面清 0。

**、 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？ 用户进程的页面？P271**

答：

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。

**10 、详细分析一个进程从创建、加载程序、执行、退出的全过程。P273**

答：

1. 创建进程，调用 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

**后续 27-36 页 （2019）**

