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。

2、 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。

3、 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。

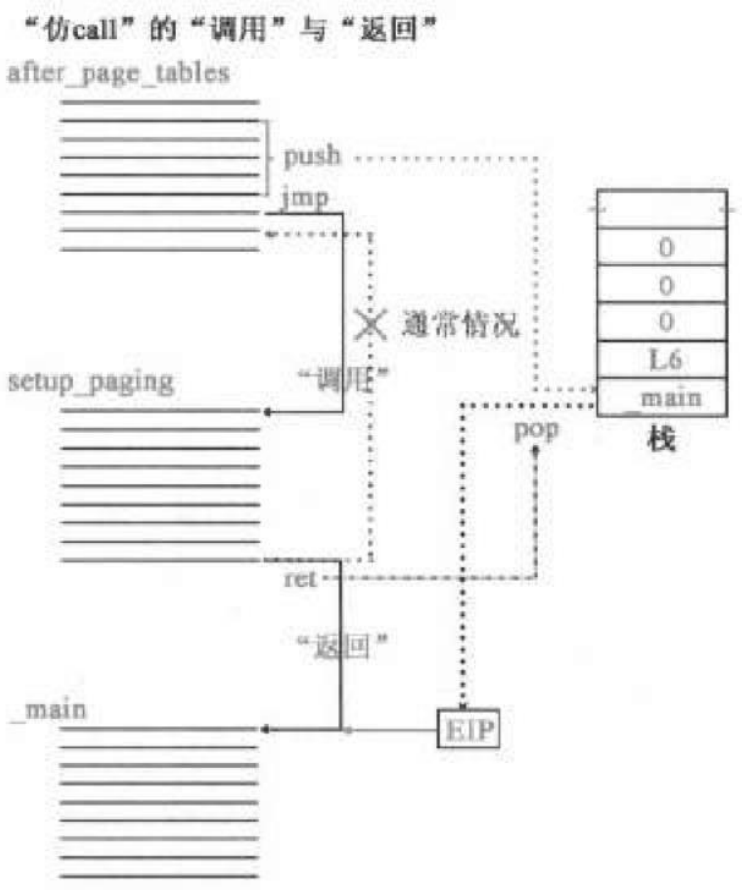
5、Linux 是用 C 语言写的，为什么没有从 main 还是开始，而是先运行 3 个汇编程序，道理何在？

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：

```
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。
```

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？

原来 GDT 所在的位置是设计代码时在 setup.s 里面设置的数据，将来这个 setup 模块所在的内存位置会在设计缓冲区时被覆盖。如果不改变位置，将来 GDT 的内容肯定会被缓冲区覆盖掉，从而影响系统的运行。这样一来，将来整个内存中唯一安全的地方就是现在 head.s 所在的位置了。

那么有没有可能在执行 setup 程序时直接把 GDT 的内容复制到 head.s 所在的位置呢？肯定不能。如果先复制 GDT 的内容，后移动 system 模块，它就会被后者覆盖；如果先移动 system 模块，后复制 GDT 的内容，它又会把 head.s 对应的程序覆盖，而这时 head.s 还没有执行。所以，无论如何，都要重新建立 GDT。

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？

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

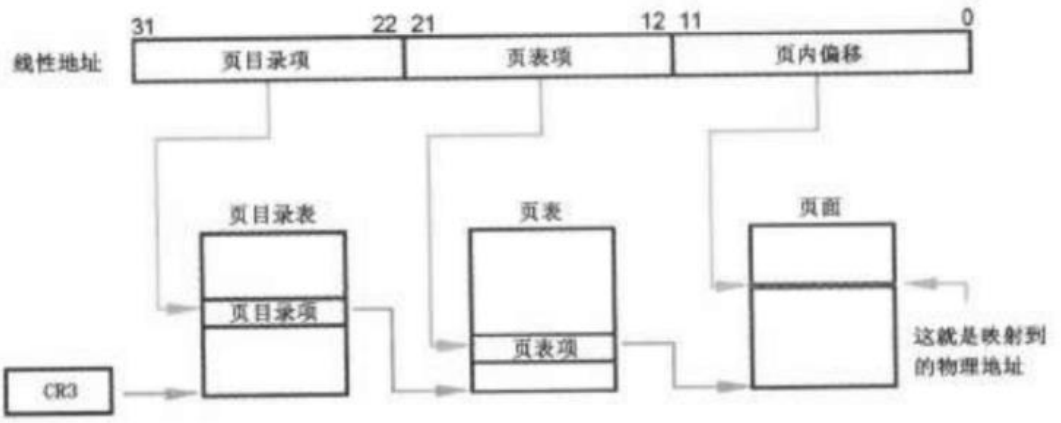


图 3-9　线性地址到物理地址映射过程示意图

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？

因为 BIOS 将从 0x00000 开始的 1KB 字节（0x00000-0x003ff）构建了中断向量表，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，操作系统只能遵守这个约定，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 在 task 数组的第 0 项，是操作系统设计者事先进程 0 的 task_struct 位于内核数据区，存储在 user_stack 中。因为在进程 0 未激活之前，使用的是 boot 阶段的 user_stack，因此位于内核数据区。

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)

```
/*进程0的task_struct
 *  INIT_TASK is used to set up the first task table, touch at
 * your own risk!. Base=0, limit=0x9ffff (=640kB)
 */
#define INIT_TASK \
/* state etc */    { 0,15,15, \    //就绪态，15个时间片
/* signals */      0,{{},},0, \
/* ec,brk... */    0,0,0,0,0,0, \
/* pid etc.. */    0,-1,0,0,0, \  //进程号0
/* uid etc */      0,0,0,0,0,0, \
/* alarm */        0,0,0,0,0,0, \
/* math */ 0, \
/* fs info */      -1,0022,NULL,NULL,NULL,0, \
/* filp */  {NULL,}, \
    { \
        {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, \          //eflags的值，决定了cli这类指令只能在0特权级使用
     0,0,0x17,0x17,0x17,0x17,0x17,0x17, \
     _LDT(0),0x80000000, \
        {} \
    }, \
}
```

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

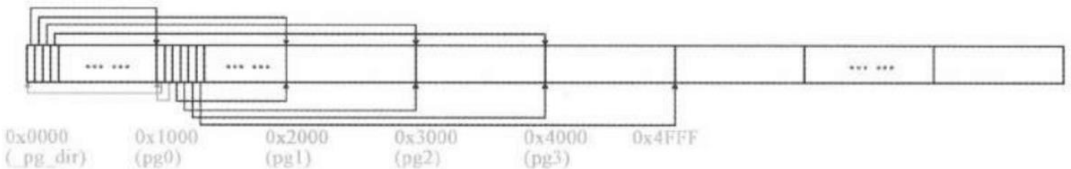


图 1-42　总体效果图

代码证据

```
// 代码路径：boot/head.s
    ...
.align 2
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
/* 下面几行中的7应看成二进制的111，是页属性，代表u/s、r/w、present，
111代表：用户u、读写rw、存在p，000代表：内核s、只读r、不存在  */
    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
    ...
```

P39 最下面

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。

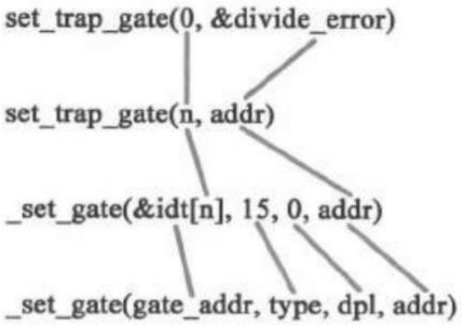


图 2-9　参数对应示意图

（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。

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。

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

```
#define INIT_TASK \
/* state etc */    { 0,15,15, \    //就绪态，15个时间片
/* signals */      0,{{},},0, \
/* ec,brk... */    0,0,0,0,0,0, \
/* pid etc.. */    0,-1,0,0,0, \   //进程号0
/* uid etc */      0,0,0,0,0,0, \
/* alarm */        0,0,0,0,0,0, \
/* math */ 0, \
/* fs info */      -1,0022,NULL,NULL,NULL,0, \
/* filp */  {NULL,}, \
    { \
        {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, \          //eflags的值，决定了cli这类指令只能在0特权级使用
     0,0,0x17,0x17,0x17,0x17,0x17,0x17, \
     _LDT(0),0x80000000, \
        {} \
    }, \
}
```

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。

5、 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。

6、 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。

7、 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

```
// 代码路径：mm/memory.c:
unsigned long get_free_page(void)       //遍历 mem map[]，找到主内存中（从高地址开始）第一个空闲页面
{                                       //参看前面的嵌入汇编的代码注释
register unsigned long __res asm("ax");
```



```
__asm__("std;repne;scasb\n\t"                   //反向扫描串（mem map[]），al（0）与 di 不等则
                                                //重复（找引用对数为 0 的项）
        "jne 1f\n\t"                            //找不到空闲页，跳转到 1
        "movb $1,1(%%edi)\n\t"                  //将 1 赋给 edi + 1 的位置，在 mem map[] 中，
                                                //将找到 0 的项的引用计数置为 1
        "sall $12,%%ecx\n\t"                    // ecx 算数左移 12 位，页的相对地址
        "addl %2,%%ecx\n\t"                     // LOW MEN + ecx，页的物理地址
        "movl %%ecx,%%edx\n\t"
        "movl $1024,%%ecx\n\t"
        "leal 4092(%%edx),%%edi\n\t"            //将 edx + 4 KB 的有效地址赋给 edi
        "rep;stosl\n\t"                         //将 eax（即"0"(0)）赋给 edi 指向的地址，目的是页面清零
        "movl %%edx,%%eax\n"
        "1:"
        :"=a" (__res)
        :"0" (0),"i" (LOW_MEM),"c" (PAGING_PAGES),
        "D" (mem_map + PAGING_PAGES-1)          //edx，mem map[] 的最后一个元素
        :"di","cx","dx");                       //第三个冒号后是程序中改变过的量
return __res;
}
```

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）；

③ 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。

8、 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。

以进程 0 创建进程 1 为例。进程 0 进入 copy_page_tables（）函数后，先为新的页表申请一个空闲页面，并把进程 0 中第一个页表里面前 160 个页表项复制到这个页面中（1 个页表项控制 1 个页面 4KB 内存空间，160 个页表项可以控制 640KB 内存空间）。进程 0 和进程 1 的页表暂时都指向了相同的页面，意味着进程 1 也可以操作进程 0 的页面。之后对进程 1 的页目录表进行设置。最后，用重置 CR3 的方法刷新页变换高速缓存。进程 1 的页表和页目录表设置完毕。 （代码见 P97-P98）

9、 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

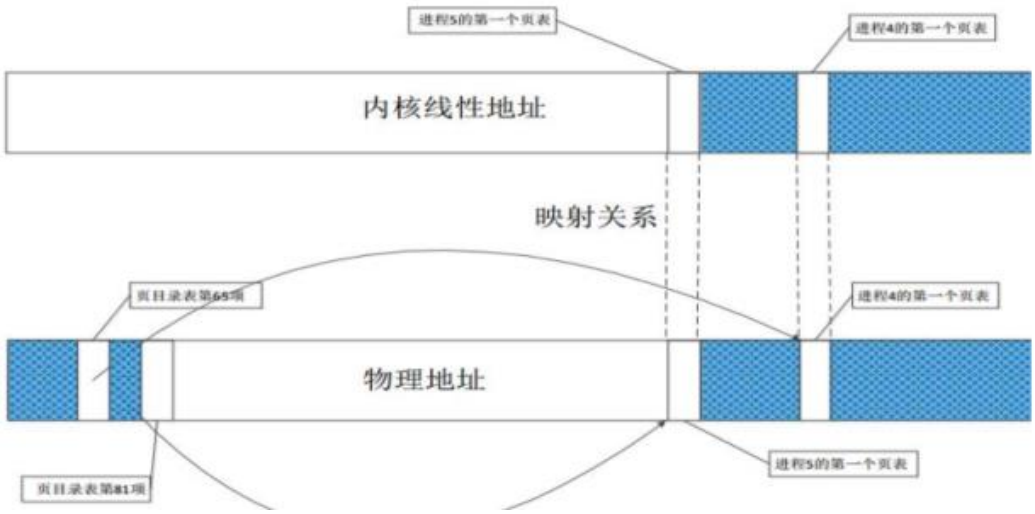
理解：内核的线性地址空间为 0x00000~0xfffff（16M），且线性地址与物理地址一一对应。为进程 1 分配的这两个页，在 16MB 的顶端倒数第一页、第二页，因此占用内核线性地址空间。

进程 0 的线性地址空间是内存的前 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（前 160 项），而这 160 个页表项即为内核第一页表的前 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面。

10、 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。

这两个页表均在内核的线性地址空间。既然是内核线性地址空间，则与物理地址空间为一一

对应关系。根据每个进程占用 16 个页目录表项，则进程 4 占用从第 64～79 项的页目录表项。同理，进程 5 占用第 80～95 项的页目录表项。由于目前只分配了一个页面（用做进程的第一个页表），则分别只需要使用第一个页目录表项即可。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。**

示意图如下所示：

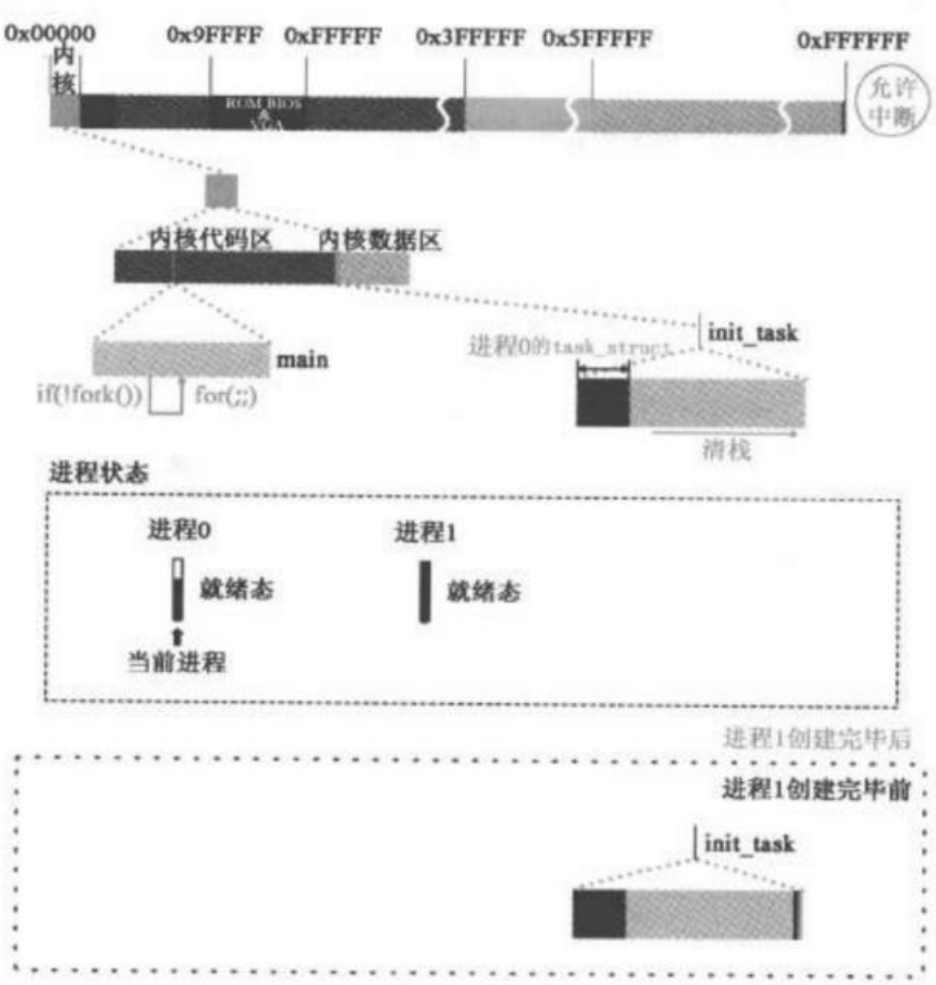


图 3-12　操作系统如何区分进程 0 与进程 1

主要涉及的代码位置如下：

Init/main.c 代码中 P103 —— if 判断

```
// 代码路径：init/main.c:
    ...
void main(void)
{
    sti();
    move_to_user_mode();
    if (!fork()) {//fork 的返回值为 1，if（！1）为假 /* we count on this going ok */
        init();   // 不会执行这一行
    }
    ...
    for(;;) pause();       //执行这一行！
}
```

Include/unistd.h 中 P102 —— fork 函数代码

```
// 代码路径：include/unistd.h:
int fork(void)
{
    long __res;
__asm__ volatile ("int $0x80"
    : "=a" (__res)          //__res 的值就是 eax，是 copy_process() 的返回值 last_pid (1)
    : "0" (__NR_ fork));
    if(__res >= 0)          //iret 后，执行这一行！ __res 就是 eax，值是 1
    return (int) __res;     //返回 1！
    errno= -__res;
    return -1;
}
```

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;

…

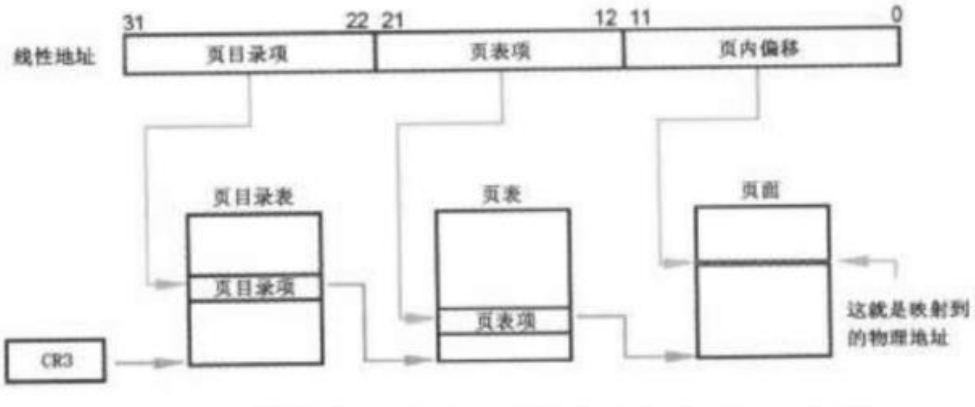
```
p->state = TASK_RUNNING;

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）

答：在 Linux 0.11 中采用了基于优先级排队的调度策略。

调度程序

schedule()函数首先扫描任务数组。通过比较每个就绪态任务的运行时间，counter 的值来确定当前哪个进程运行的时间最少。哪一个的值大，就表示运行时间还不长，于是就选中该进程，并使用任务切换宏函数切换到该进程运行。

如果此时所有处于就绪状态进程的时间片都已经用完，系统就会根据每个进程的优先 权值 priority，对系统中所有进程（包括正在睡眠的进程）重新计兑每个任务需要运行的时间片值 counter。 计算的公式是：

Counter=counter /2+priority

这样对于正在睡眠的进程当它们被唤醒时就具冇较高的时间片 counter 值.然后 schedule0 函数重新扫描 任务数组屮所有处就绪状态的进程，并重复上述过程，直到选择出一个进程为止。最后调用 switch_to()执行实际的进程切换操作。

如果此时没有其他进程可运行，系统就会选择进程 0 运行。对于 Linux 0.11 来说，进程 0 会调用 pause() 把自己置为可中断的睡眠状态并冉次调用 schedule0。不过在调度进程运行时，schedule0 并不在意进程 0 处于什么状态。只要系统空闲就调度进程 0 运行。

还有一种答案：

1. 进程中有就绪进程，且时间片没有用完。

正常情况下，schedule()函数首先扫描任务数组。通过比较每个就绪（TASK_RUNNING）任务的运行时间递减滴答计数 counter 的值来确定当前哪个进程运行的时间最少。哪一个的值大，就表示运行时间还不长，于是就选中该进程，最后调用 switch_to()执行实际的进程切换操作

2. 进程中有就绪进程，但所有就绪进程时间片都用完（c=0）如果此时所有处于 TASK_RUNNING 状态进程的时间片都已经用完，系统就会根据每个进程的优先权值 priority，对系统中所有进程（包括正在睡眠的进程）重新计算每个任务需要运行的时间片值 counter。计算的公式是：

counter = counter + priority/2

然后 schdeule()函数重新扫描任务数组中所有处于 TASK_RUNNING 状态，重复上述过程，直到选择出一个进程为止。最后调用 switch_to()执行实际的进程切换操作。

3. 所有进程都不是就绪的 c=-1

此时代码中的 c=-1，next=0，跳出循环后，执行 switch_to(0)，切换到进程 0 执行，因此所有进程都不是就绪的时候进程 0 执行。

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？

检查设备是否正忙，若目前该设备没有请求项，本次是唯一一个请求，之前无链表，则将该设备当前请求项指针直接指向该请求项，作为链表的表头。并立即执行相应设备的请求函数。

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。do_hd_request()函数主要用于处理当前硬盘请求项。但其中的 dev 含义并不一致。“dev = MINOR(CURRENT->dev);”表示取设备号中的子设备号。“dev /= 5;”此时，dev 代表硬盘号（硬盘 0 还是硬盘 1）。由于 1 个硬盘中可以存在 1--4 个分区，因此硬盘还依据分区的不同用次设备号进行指定分区。因此硬盘的逻辑设备号由以下方式构成

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;
        }
```

如果 if 语句判断为真，则请求项对应的缓冲块数据没有读完(nr_sectors>0)，“—CURRENT->nr_sectors”将递减请求项所需读取的扇区数值。若递减后不等于 0，表示本项请求还有数据没读完内核将再次把 read_intr()绑定在硬盘中断服务程序上，以待硬盘在读出另 1 扇区数据后发出中断并再次调用本函数，之后中断服务程序返回。<1 个块两个扇区，读两次>

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？

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。
我们需要使用 inline 函数 - 从内核空间创建进程(forking)将导致没有写时复制（COPY ONWRITE）！直到执行一个 execve 调用。这对堆栈可能带来问题。处理的方法是在 fork()调用之后不让 main()使用任何堆栈。因此就不能有函数调用 - 这意味着 fork 也要使用 inline 代码，否则我们在从 fork()退出时就要使用堆栈了。实际上只有 pause

和 fork 需要 inline 方式，以保证从 main()中不会弄乱堆栈，但是我们同时还定义了其它一些函数

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。

```
int fork(void)                          // 参看 2.5 节、2.9 节、2.14 节有关嵌入汇编的代码注释
{
long __res;
__asm__ volatile ("int $0x80"           // int 0x80 是所有系统调用函数的总入口，fork() 是其中
                                        // 之一，参看 2.9 节的讲解及代码注释
    : "=a" (__res)                      // 第一个冒号后是输出部分，将 _res 赋给 eax
    : "0" (__NR_ fork));                // 第二个冒号后是输入部分，"0"：同上寄存器，即 eax，
                                        // __NR_ fork 就是 2，将 2 给 eax
    if (__res >= 0)                     // int 0x80 中断返回后，将执行这一句
    return (int) __res;
    errno= -__res;
    return -1;
}
```

```
_system_call:                           # int 0x80——系统调用的总入口
    cmpl $nr_system_calls-1,%eax
    ja bad_sys_call
    push %ds                            # 下面 6 个 push 都是为了 copy_process() 的参数，请记住
                                        # 压栈的顺序，别忘了前面的 int 0x80 还压了 5 个寄存器的值进栈
    push %es
    push %fs
    pushl %edx
    pushl %ecx                          # push %ebx,%ecx,%edx as parameters
    pushl %ebx                          # to the system call
    movl $0x10,%edx                     # set up ds,es to kernel space
    mov %dx,%ds
    mov %dx,%es
    movl $0x17,%edx                     # fs points to local data space
    mov %dx,%fs
    call _sys_call_table(,%eax,4)       # eax 是 2，可以看成 call (_sys_call_table + 2×4) 就是
                                        # _sys_fork 的入口
    pushl %eax
    movl _current,%eax
    cmpl $0,state(%eax)                 # state
```

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。

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 ？

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。具体而言，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）

13.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

后续 2018 年 思考题 8-14