# 思考讨论

## 1.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？

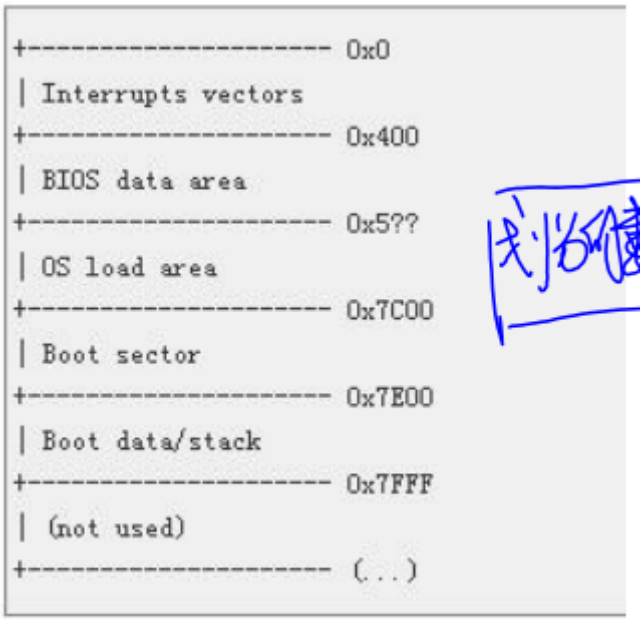
> 计算机加电启动的时候，操作系统并没有在内存中，*CPU* 也不能从外设运行操作系统，所以必须将操作系统加载到内存中，该过程的最开始部分是由 *BIOS* 完成的。在加电后，*BIOS* 需要完成一些检测工作，设置实模式下的中断向量表和服务程序，并将操作系统的引导扇区加载至 *0x7C00* 处，然后将跳转至 *0x7C00* 运行操作系统自身的代码。所以计算机启动最开始运行的是 *BIOS* 代码。

## 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分属不同的团队，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。

## 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？

> 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。




## 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;`

*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`lidt idt_48; lgdt gdt_48`。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。

## 5.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，究竟有什么内涵？

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。

## 6.保护模式在“保护”什么？它的“保护”体现在哪里？特权级的目的和意义是什么？分页有“保护”作用吗？

> （*1*） 保护模式在“保护”什么？它的“保护”体现在哪里？
>
> 保护操作系统的安全，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：
>
> - 在 *GDT*、 *LDT* 及 *IDT* 中，均有对应界限、特权级等；
> - 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；
> - 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。
>
> （*2*）特权级的目的和意义是什么？
>
> 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。
>
> （*3*）分页有“保护”作用吗？
>
> 分页机制有保护作用，使得用户进程不能直接访问内核地址，进程间也不能相互访问。用户进程只能使用逻辑地址，而逻辑地址通过内核转化为线性地址，根据内核提供的专门为进程设计的分页方案，由*MMU*非直接映射转化为实际物理地址形成保护。此外，通过分页机制，每个进程都有自己的专属页表，有利于更安全、高效的使用内存，保护每个进程的地址空间。
>
> 为什么特权级是基于段的？（超纲备用）

> 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。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，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。

## 7.在**setup**程序里曾经设置过**gdt**，为什么在**head**程序中将其废弃，又重新设置了一个？为什么设置两次，而不是一次搞好？

（P33-点评）

> 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*。

## 8.进程**0**的**task_struct**在哪？具体内容是什么？

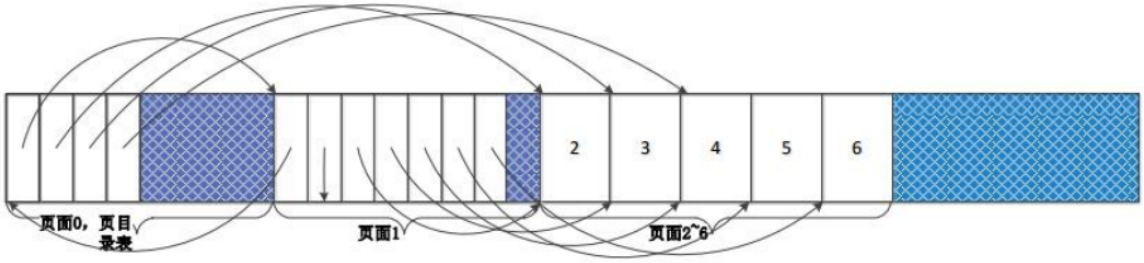
> 进程 *0* 的 *task_struct* 位于内核数据区，因为在进程 *0* 未激活之前 ，使用的是 *boot*阶段的 *user_stack*，因此存储在 *user_stack* 中。
>
> 具体内容如下：
>
> 包含了进程 *0* 的进程状态、进程 *0* 的 *LDT*、进程 *0* 的 *TSS* 等等。其中 *ldt* 设置了代码段和堆栈段的基址和限长*(640KB)*，而 *TSS* 则保存了各种寄存器的值，包括各个段选择符。
>
> 代码如下：*(P68，若未要求没时间可不写)*

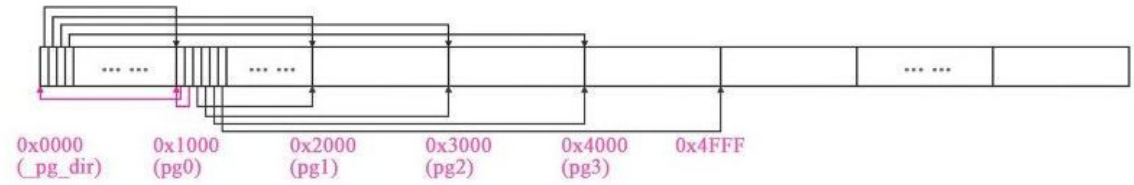
```
//进程0的task_struct的值
/*
 *  INIT_TASK is used to set up the first task table, touch
at
 * your own risk!. Base=0, limit=0x9ffff (=640kB)
 */
#define INIT_TASK \
/* state etc */ { 0,15,15, \
/* signals */   0,{{},},0, \
```

```
/* ec,brk... */ 0,0,0,0,0,0, \
/* pid etc.. */ 0,-1,0,0,0, \
/* 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, \
     0,0,0x17,0x17,0x17,0x17,0x17,0x17, \
     _LDT(0),0x80000000, \
        {} \
    }, \
}
```

**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（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*））




代码证据（*P39*）

```
//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
    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
    xorl %eax,%eax      /* pg_dir is at 0x0000 */
    movl %eax,%cr3      /* cr3 - page directory start */
    movl %cr0,%eax
    orl $0x80000000,%eax
```

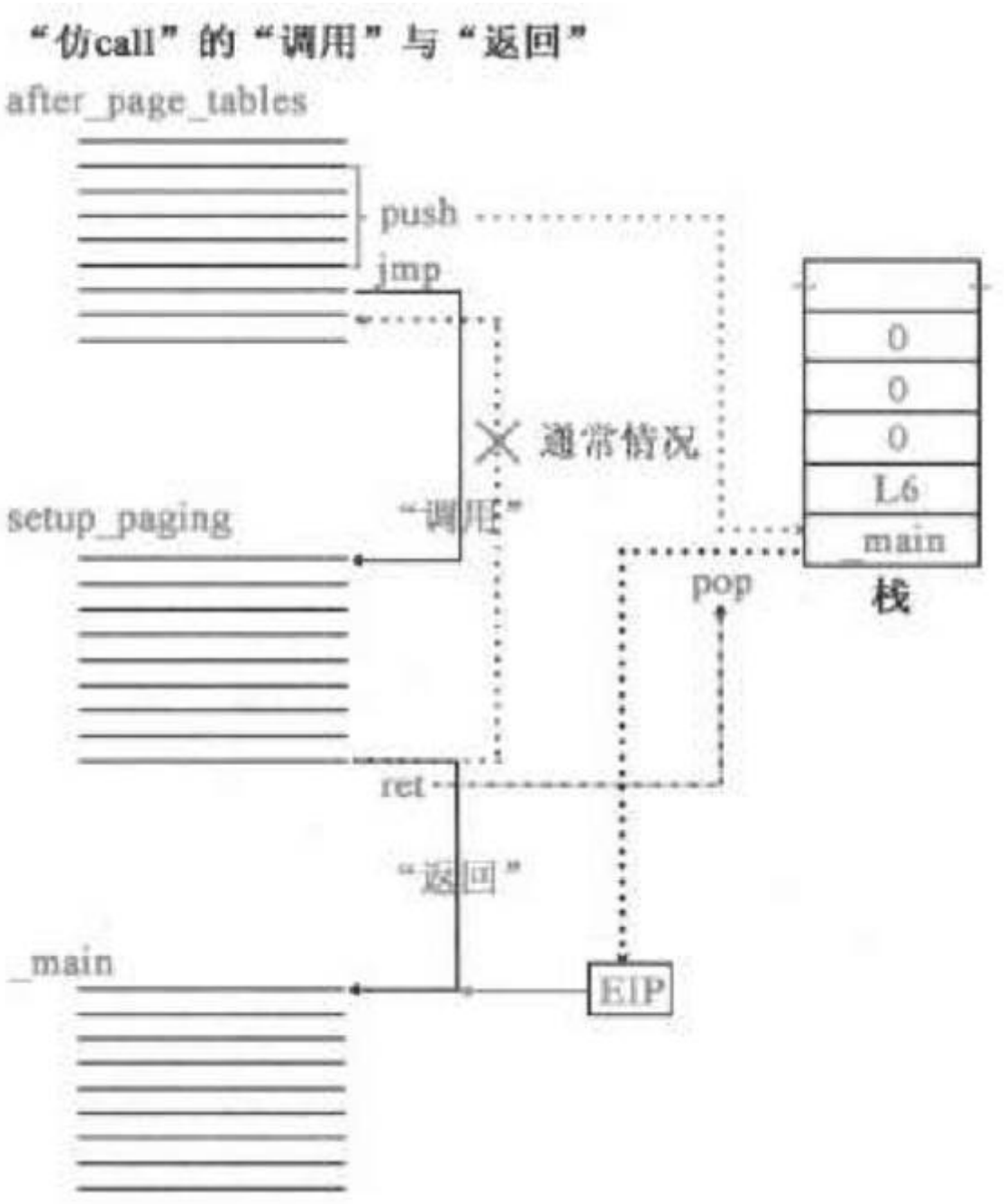
```
        movl %eax,%cr0          /* set paging (PG) bit */
        ret             /* this also flushes prefetch-queue */
```

## 10.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？

> 在 *idt* 前面有 *184* 个字节（*0x5400-0x54b7*）的剩余代码，包含了 *after_page_tables*、 *ignore_int* 和*setup_paging*代码段，其中 *after_page_tables* 往栈中压入了些参数，*ignore_int* 用作初始化中断时的中断处理函数，*setup_paging* 则是初始化分页。
>
> 剩余的原因：
>
> *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: main*，以使得系统可能出错时，返回到 *L6* 处执行。
>
> *ignore_int* 为中断处理函数，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，用于分页，在该函数中对 *0x0000* 和 *0x5000* 的进行了初始化操作。该代码需要“剩余”用于跳转到 *main*，即执行“*ret*”指令。

## 11.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。
```

```
//代码路径：boot\\head.s
after_page_tables:
    pushl $0        # These are the parameters to main :-)
    pushl $0
    pushl $0
    pushl $L6       # return address for main, if it decides
to.
    pushl $_main
    jmp setup_paging
L6:
    jmp L6          # main should never return here, but
                # just in case, we know what happens.
```

**12.**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。

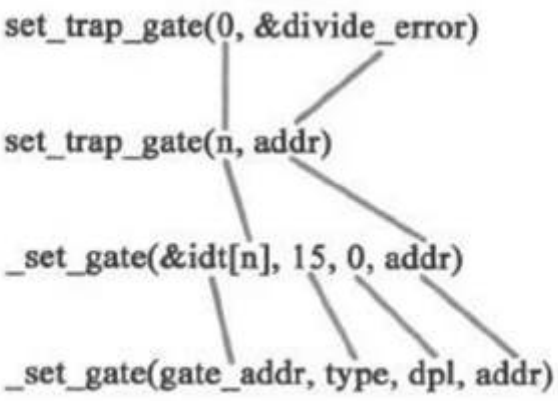



图 2-9　参数对应示意图

如上所示，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。

```
//代码路径：include\asm\system.h
#define set_trap_gate(n,addr) \
    _set_gate(&idt[n],15,0,addr)
#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))
```

刚开始中断服务程序地址*&divide_error* 存储在*edx*中，段选择子*0008*存储在*eax*的高字，将*edx*低字赋给*eax*的低字组成中断描述符表的低*32*位，*0x8000+*（*dpl<<13*）+（*type<<8*）即二进制*1000111100000000*是对*p*位、*type*位、*dpl*位的设置，然后赋给*edx*的低位组成中断描述符表的高*32*位，最后将*edx*、*eax*分别写入中断描述符表的高*32*位和低*32*位。

## 13.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

```
//代码路径：include\asm\system.h
#define move_to_user_mode() \

__asm__(“movl %%esp, %%eax\n\t” \

……

“pushfl\n\t” \ // EFLAGS 进栈

……

”)
```

第二部分代码见 *P 68 INIT_TASK* 的 *tss* 部分的设置中

```
//进程0的task_struct的值
/*
 *  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, \
/* signals */   0,{{},},0, \
/* ec,brk... */ 0,0,0,0,0,0, \
/* pid etc.. */ 0,-1,0,0,0, \
/* 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, \
     0,0,0x17,0x17,0x17,0x17,0x17,0x17, \
     _LDT(0),0x80000000, \
```

```
        {} \
    }, \
}
```

而进程*1*在*copy_process*中*TSS*里，设置了*EFLAGS*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*。

```
//代码路径：kernel\fork.c
int copy_process(int nr,long ebp,long edi,long esi,long
gs,long none,
        long ebx,long ecx,long edx,
        long fs,long es,long ds,
        long eip,long cs,long eflags,long esp,long ss)
{
    //…
    p->tss.eip = eip;
    p->tss.eflags = eflags;
    p->tss.eax = 0;
    //…
    return last_pid;
}
```

## 14.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，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（ *P 68*）：

```
//进程0的task_struct的值
/*
 *  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, \
/* signals */   0,{{},},0, \
/* ec,brk... */ 0,0,0,0,0,0, \
/* pid etc.. */ 0,-1,0,0,0, \
/* 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, \
     0,0,0x17,0x17,0x17,0x17,0x17,0x17, \
     _LDT(0),0x80000000, \
        {} \
    }, \
}
```

## 15.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。

> *set_trap_gate* 和*set_intr_gate*的*dpl*是*3*，*set_system_gate*的*dpl*是*0*。*dpl*为*0*表示只能在内核态下允许，*dpl*为*3*表示系统调用可以由*3*特权级调用。
>
> 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*。

## 16.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。
>
> 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。
>
> 设计者通过代码模拟 *int*（中断） 压栈， 当执行 *iret* 指令时， 硬件 自动将 *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。

## 17.在Linux操作系统中大量使用了中断、异常类的处理，究竟有什么好处？

> *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。在未引入中断、异常处理类处理理念以前，*CPU*采用主动轮询”的方式来处理硬件信号，下降了系统的综合效率。

> 因此，现代操作系统采用以“被动模式”代替“主动轮询”模式来处理中断问题。这种 *32* 位中断服务体系是为适应一种被动相应中断信号的机制而建立的。通过大量使用中断、异常类的处理， *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“被动响应”模式替代“主动轮询”模式来处理中断问题是现代操作系统之所以称之为“现代”的一个重要标志。

## 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。

> 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。

## 19.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。

```
//代码路径：mm\memory.c
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:"
    :"=a" (__res)
```

```
        :"0" (0),"i" (LOW_MEM),"c" (PAGING_PAGES),
        "D" (mem_map+PAGING_PAGES-1)
        :"di","cx","dx");
    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。

## 20.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。

进程*1*此时是一个空架子，尚未对应的程序，它的页表又是从进程*0*的页表复制过来的，它们管理的页面彻底一致，也就是它暂时和进程*0*共享一套页面管理结构。

参考代码（时间不够可以不写）：

```
//代码路径：kernel/fork.c
int copy_mem(int nr,struct task_struct * p)
{
    ......
    set_base(p->ldt[1],new_code_base);//设置子进程代码段基址
    set_base(p->ldt[2],new_data_base);//设置子进程数据段基址
    //为进程1创建第一个页表、复制进程0的页表，设置进程1的页目录项
    if
(copy_page_tables(old_data_base,new_data_base,data_limit)) {
```

```
            free_page_tables(new_data_base,data_limit);
            return -ENOMEM;
        }
        return 0;
}

//代码路径：mm/memory.c
......
#define invalidate（）\
__asm__（"movl%%eax，%%cr3"："a"（0））//重置CR3为0
......
int copy_page_tables(unsigned long from,unsigned long to,long
size)
{
    unsigned long * from_page_table;
    unsigned long * to_page_table;
    unsigned long this_page;
    unsigned long * from_dir, * to_dir;
    unsigned long nr;
/*0x3fffff是4 MB，是一个页表的管辖范围，二进制是22个1，||的两边必须同为
0，所以，from和to后22位必须都为0，即4 MB的整数倍，意思是一个页表对应4
MB连续的线性地址空间必须是从0x000000开始的4 MB的整数倍的线性地址，不能是
任意地址开始的4 MB，才符合分页的要求*/
    if ((from&0x3fffff) || (to&0x3fffff))
        panic("copy_page_tables called with wrong
alignment");
/*一个页目录项的管理范围是4 MB，一项是4字节，项的地址就是项数x4，也就是项
管理的线性地址起始地址的M数，比如：0项的地址是0，管理范围是0～4 MB，1项的
地址是4，管理范围是4～8 MB，2项的地址是8，管理范围是8～12MB……>>20就是
地址的MB数，&0xffc就是&111111111100b，就是4 MB以下部分清零的地址的
MB数，也就是页目录项的地址*/
    from_dir = (unsigned long *) ((from>>20) & 0xffc); /*
_pg_dir = 0 */
    to_dir = (unsigned long *) ((to>>20) & 0xffc);
    size = ((unsigned) (size+0x3fffff)) >> 22;
    for( ; size-->0 ; from_dir++,to_dir++) {
        if (1 & *to_dir)
            panic("copy_page_tables: already exist");
        if (!(1 & *from_dir))
            continue;
        from_page_table = (unsigned long *) (0xfffff000 &
*from_dir);
```

```
        if (!(to_page_table = (unsigned long *)
get_free_page()))
            return -1;  /* Out of memory, see freeing */
        *to_dir = ((unsigned long) to_page_table) | 7;
        nr = (from==0)?0xA0:1024;
        for ( ; nr-- > 0 ; from_page_table++,to_page_table++)
{
            this_page = *from_page_table;
            if (!(1 & this_page))
                continue;
            this_page &= ~2;
            *to_page_table = this_page;
            if (this_page > LOW_MEM) {
                *from_page_table = this_page;
                this_page -= LOW_MEM;
                this_page >>= 12;
                mem_map[this_page]++;
            }
        }
    }
    invalidate();
    return 0;
}
```

**21.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。**

> 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。

进程*0*创建进程*1*的时候，先后通过*get_free_page*函数从物理地址中取出了两个页，但是并没有将这两个页的物理地址填入任何新的页表项中。此时只有内核的页表中包含了与这段物理地址对应的项，也就是说此时只有内核页表中有页表项指向这两个页的首地址，所以这两个页占用了内核线性空间。

**22.**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。根据每一个进程占用*16*个页目录表项，则进程*4*占用从第*65*～*81*项的页目录表项。同理，进程*5*占用第*81*～*96*项的页目录表项。因为目前只分配了一个页面（用作进程的第一个页表），则分别只须要使用第一个页目录表项便可。映射关系如图：

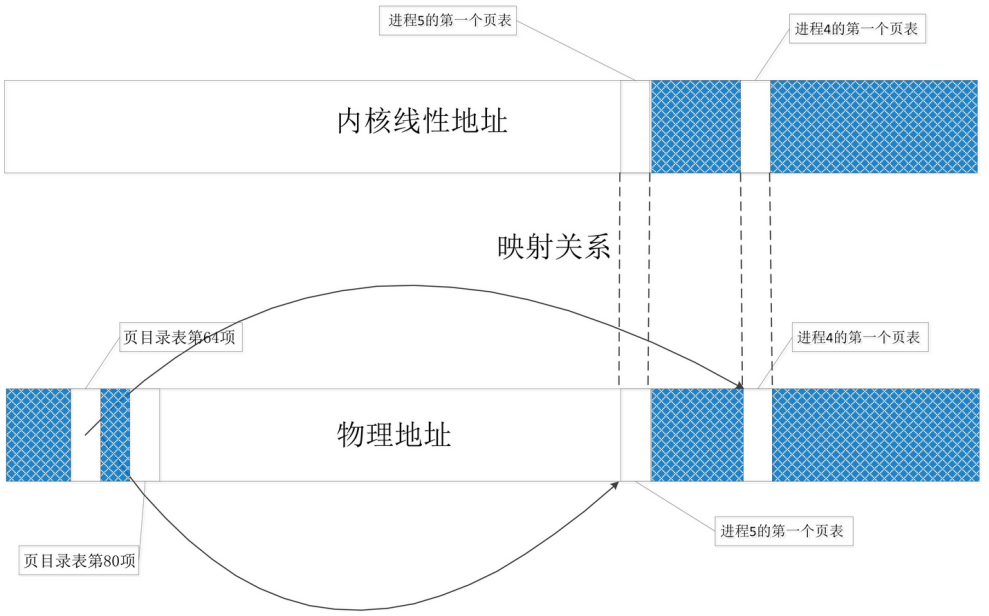



图 1 线性地址及物理地址映射关系

**23.**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。

```
#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]));
}
```

> `ljmp %0\n\t`经过任务门机制并未实际使用任务门，将*CPU*的各个寄存器值保存在进程*0*的*TSS*中，将进程*1*的*TSS*数据以*LDT*的代码段、数据段描述符数据恢复给*CPU*的各个寄存器，实现从*0*特权级的内核代码切换到*3*特权级的进程*1*代码执行。其中*tss.eip*也天然恢复给了*CPU*，此时*EIP*指向的就是*fork*中的*if(__res >= 0)*语句。
>
> 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。

## 24.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。

> 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。
>
> 主要涉及的代码位置如下：

*Init/main.c* 代码中 *P103 —— if* 判断

```
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

```
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_##name));
if (__res >= 0) //iret后，执行这一行！__res就是eax，值是1
    return (type) __res; //返回1!
errno = -__res;
return -1;
}
```

```
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;
}
```

*Kernel/sched.c* 中 *P105——sys_pause* 和 *schedule* 函数代码

```
// 代码路径：kernel/sched.c:
int sys_pause(void)
{
// 将进程 0 设置为可中断等待状态，如果产生某种中断，或其他进程给这个进程发送特定信号…才有可能将
// 这个进程的状态改为就绪态
        current->state= TASK_INTERRUPTIBLE;
        schedule();
        return 0;
}
```

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;
```

## 25、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）

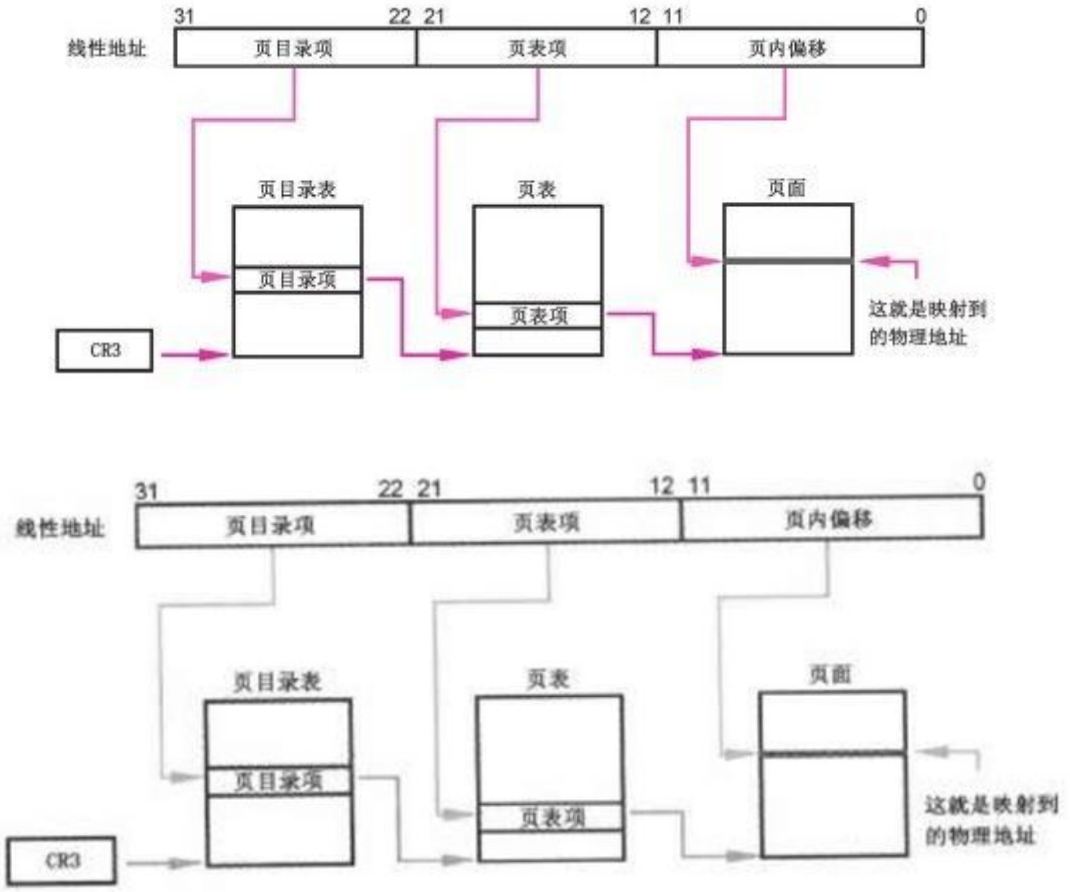



图 3-9　线性地址到物理地址映射过程示意图

## 26、getblk函数中，申请空闲缓冲块的标准就是b_count为0，而申请到之后，为什么在wait_on_buffer(bh)后又执行if（bh->b_count）来判断b_count是否为0？

> 字段*b_count*，用来标记“每个缓冲块有多少个进程在共享”。只有当*b_count=0*时，该缓冲块才能被再次分配。
>
> 因为 *wait_on_buffer(bh)*函数中有睡眠，若在睡眠等待的过程中，该缓冲块又被其他进程占用，那么就要再重头开始搜索缓冲块。若没被占用则判断该缓冲块是否已被修改过，但被修改过，则将该块写盘，并等待该块解锁。此时如果该缓冲块又被别的进程占用，那么又要 *repeat*。所以又要执行 *if*（*bh->b_count*）
>
> 可能引发异常例子（时间不够可以不写）：
>
> 每个缓冲块有一个进程等待队列，假设此时*B*、*C*两进程在队列中，当该缓冲块被解锁时，进程*C*被唤醒（它开始使用缓冲区之前需先唤醒进程*B*，使进程*B*从挂起进入就绪状态），将缓冲区加锁，一段时间后，进程*C*又被挂起，但此时缓冲区进程*C*任在使用。这时候，进程*B*被调度，*“if (bh->b_count)”*该缓冲区任是加锁状态，进程*B*重新选择缓冲区...如果，不执行该判断将造成进程*B*操作一个被加锁的缓冲区，引发异常。

## 27、b_dirt已经被置为1的缓冲块，同步前能够被进程继续读、写？给出代码证据。

> 该块同步前，能够被进程继续读、写
>
> 缓冲块是否能被进程读写，取决于*b_uptodate*。*b_uptodate* 设置为 *1* 后，内核就可以支持进程共享该缓冲块的数据了，读写都可以，读操作不会改变缓冲块的内容，所以不影响数据，而执行写操作后，就改变了缓冲块的内容，就要将 *b_dirt*标志设置为 *1*。由于此前缓冲块中的数据已经用硬盘数据块更新了，所以后续的同步未被改写的部分不受影响，同步后的结果是进程希望的，同步是不更改缓冲块中数据的，所以 *b_uptodate*仍为 *1*。即进程在 *b_dirt* 置为 *1* 时，仍能对缓冲区数据进行读写。读写文件和获取缓冲块均与 *b_dirt* 没有任何关系
>
> 代码
>
> *P331*的*file_write*
>
> *P314*的 *file_read*
>
> *P330*的 *bread getblk*

## 28、分析**panic**函数的源代码，根据你学过的操作系统知识，完整、准确的判断**panic**函数所起的作用。假如操作系统设计为支持内核进程（始终运行在**0**特权级的进程），你将如何改进**panic**函数？

> *panic()*函数是当系统发现无法继续运行下去的故障时将调用它，会导致程序终止，然后由系统显示错误号。如果出现错误的函数不是进程 *0*，那么就要进行数据同步，把缓冲区中的数据尽量同步到硬盘上。遵循了 *Linux* 尽量简明的原则。关键字 *volatile* 用于告诉 *gcc* 该函数不会返回，死机
>
> 改进 *panic* 函数：将死循环 *for(;;)*；改进为跳转到内核进程（始终运行在 *0* 特权级的进程），让内核继续执行。

```c
//代码路径：kernel/panic.c
#include <linux/kernel.h>
#include <linux/sched.h>
void sys_sync(void);    /* it's really int */
volatile void panic(const char * s)
{
    printk("Kernel panic: %s\n\r",s);
    if (current == task[0])
        printk("In swapper task - not syncing\n\r");
    else
        sys_sync();
    for(;;);
}
```

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

**29、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有较高的时间片 counter 值.然后 scheduleO 函数重新扫描 任务数组中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 scheduleO。不过在调度进程运行时，scheduleO 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。

> 另一种答案：
>
> 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。

## 30、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。

## 31、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。

这个答案最优质：

答：*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。

> 补充答案

**答：** ① （P325）b_uptodate 针对进程方向，它的作用是，高速内核，只要缓冲块的 b_uptodate 字段被设置为 1，缓冲块的数据已经是数据块中最新的，就可以放心地支持进程共享缓冲块的数据。反之，如果 b_uptodata 为 0，就是提醒内核缓冲块并没有用绑定的数据块中数据更新，不支持进程共享该缓冲块，从而保证缓冲块数据的正确性。

② 不会引起数据混乱（P329） 写哪段更好

b_uptodate 被设置为 1 后，针对该缓冲块无非会发生读写两方面情况

读情况，缓冲块是新建的，虽然里面是垃圾数据，考虑到是新建文件，这时候不存在读没有内容的文件数据块的逻辑需求，内核代码不会做出这种愚蠢的动作。

写情况，由于新建缓冲块被清零、新建的硬盘数据块都是垃圾数据，此时缓冲块和数据块里面的数据都不是进程需要的，无所谓是否更新、是否覆盖。无所谓更新，可以“等效地”看成已经更新。所以，执行写操作不会违背进程的本意。

② 补充

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

## 32、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？

> 答：检查设备是否正忙，若目前该设备没有请求项，如果指定设备*dev*当前请求项（*dev->current_request ==NULL*）为空，则表示目前设备没有请求项，本次是第*1*个请求项，也是唯一一个请求，之前无链表。因此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。

## 33、do_hd_request()函数中dev的含义始终一样吗？

题目出处代码（P122）：

```
//代码路径：kernel\blk_drv\hd.c
void do_hd_request(void)
{
    int i,r;
    unsigned int block,dev;
    ......
    dev = MINOR(CURRENT->dev);
    block = CURRENT->sector;
    if (dev >= 5*NR_HD || block+2 > hd[dev].nr_sects) {
        end_request(0);
        goto repeat;
    }
    block += hd[dev].start_sect;
    dev /= 5;
    ......
}
```

> 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*）。

## 34、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;
}
```

答案见P131页

> 答：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)*，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。“—*CURRENT->nr_sectors”*将递减请求项所需读取的扇区数值。
>
> 注：*<1*个块两个扇区，读两次>

## 35、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，直接使用。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，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。

## 36、getblk（）函数中，两次调用wait_on_buffer（）函数，两次的意思一样吗？

> *(P113+P114）*
>
> 答：一样。都是等待缓冲块解锁。第一次调用时，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。

## 37、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）

> 答：
>
> *getblk()*函数主要是获取高速缓冲中的指定缓冲块。下面的宏用于判断缓冲块的修改标志，并定义修改标志的权重比锁定标志大。
>
> `#define BADNESS(bh) (((bh)->b_dirt<<1)+(bh)->b_lock)`
>
> *Continue*：*tmp*指向的是空闲链表的第一个空闲缓冲块头*“tmp = free_list;”*。如果该缓冲块正在被使用，即*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。
>
> 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

## 38、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，根据具体的读写操作，执行 *if* 的内容说明没有找到空请求项：如果*request[32]*中没有一项是空闲的，则查看此次请求是不是提前读写，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。让本次请求先睡眠*“sleep_on(&wait_for_request);”*以等待*request*请求队列腾出空闲项，一段时间后再次搜索请求队列。

# 往年思考讨论

## 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。

## 2、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？

> 答：
>
> 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。

## 3、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？

> 答：
>
> *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。

## 4、为什么static inline _syscall0(type,name)中需要加上关键字inline？

> 答：
>
> 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。

## 5、根据代码详细说明**copy_process**函数的所有参数是如何形成的？

> 答：
>
> *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。

## 6、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**）

> 答：
>
> 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。

**7、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;
> ```

**8、用户进程自己设计一套LDT表，并与GDT挂接，是否可行，为什么？**

> 答：
>
> 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。

## 9、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*。

## **10**、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？**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。

## **11**、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*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

## 12、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。

> 答：
>
> 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。
>
> 另一种答案：
>
> 依次创建*3*个用户进程，每个进程都有自己的*task*。假设进程*1*先执行，需要压栈产生缺页中断，内核为其申请空闲物理页面，并映射到进程*1*的线性地址空间。这时产生时钟中断，轮到进程*2*执行，进程*2*也执行同样逻辑的程序。之后，又轮到进程*3*执行，也是压栈，并设置*text*。可见，三个进程虽程序相同，但数据独立，用*TSS*和*LDT*实现对进程的保护。

## 13、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？**P264,268-270**

> 答：
>
> ① 缺页中断产生 *P264*

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*P*位为*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。操作系统会调用*_do_no_page*为进程申请空闲页面，将程序加载到新分配的页面中，并建立页目录表-页表-页面的三级映射管理关系。

② 页写保护中断 *P268-270*

假设两个进程共享一个页面，该页面处于写保护状态即只读，此时若某一进程执行写操作，就会产生*“*页写保护*”*异常。操作系统会调用*_do_wp_page*，采用写时复制的策略，为该进程申请空闲页面，将该进程的页表指向新申请的页面，然后将原页表的数据复制到新页面中，同时将原页面的引用计数减*1*。该进程得到自己的页面，就可以执行写操作。

## 14、为什么要设计缓冲区，有什么好处？

答：

缓冲区的作用主要体现在两方面：

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 数据块复用，提高对块设备文件操作的运行效率。在计算机中，内存间的数据交换速度是内存与硬盘数据交换速度的*2*个量级，如果某个进程将硬盘数据读到缓冲区之后，其他进程刚好也需要读取这些数据，那么就可以直接从缓冲区中读取，比直接从硬盘读取快很多。如果缓冲区的数据能够被更多进程共享的话，计算机的整体效率就会大大提高。同样，写操作类似。

另一份答案

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。

## 15、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？

> 答：
>
> *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*指向缓冲块，用于找到缓冲块的位置。
>
> 进程与缓冲区及缓冲区与硬盘之间都是以缓冲块为单位进行数据交互的，而*b_blocknr*，*b_dev*唯一标识一个块，用于保证数据交换的正确性。另外缓冲区中的数据被越多进程共享，效率就越高，因此要让缓冲区中的数据块停留的时间尽可能久，而这正是由*b_blocknr*，*b_dev*决定的，内核在*hash*表中搜索缓冲块时，只看设备号与块号，只要缓冲块与硬盘数据的绑定关系还在，就认定数据块仍停留在缓冲块中，就可以直接用。
>
> *b_uptodate*与*b_dirt*，是为了解决缓冲块与数据块的数据正确性问题而存在的。*b_uptodate*针对进程方向，如果*b_uptodate*为*1*，说明缓冲块的数据已经是数据块中最新的，可以支持进程共享缓冲块中的数据；如果*b_uptodate*为*0*，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。
>
> *b_dirt*是针对硬盘方向的，*b_dirt*为*1*说明缓冲块的内容被进程方向的数据改写了，最终需要同步到硬盘上；*b_dirt*为*0*则说明不需要同步
>
> *b_count*记录每个缓冲块有多少进程共享。*b_count*大于*0*表明有进程在共享该缓冲块，当进程不需要共享缓冲块时，内核会解除该进程与缓冲块的关系，并将*b_count*数值减*1*，为*0*表明可以被当作新缓冲块来申请使用。
>
> *b_lock*为*1*说明缓冲块正与硬盘交互，内核会拦截进程对该缓冲块的操作，以免发生错误，交互完成后，置*0*表明进程可以操作该缓冲块。
>
> *b_wait*记录等待缓冲块的解锁而被挂起的进程，指向等待队列前面进程的*task_struct*。

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

> 答：*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。

## 17、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***

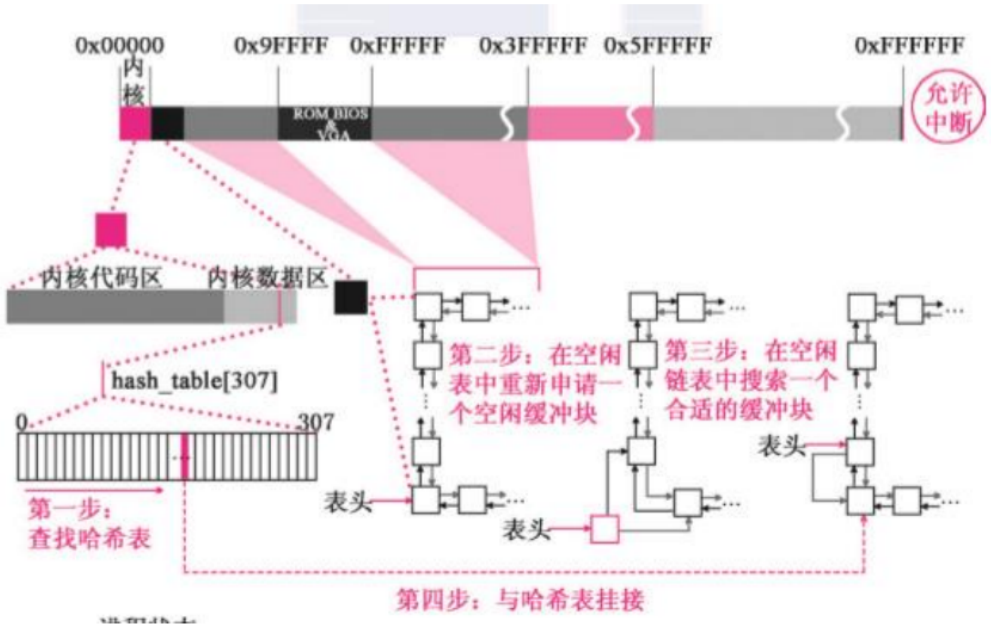
*getblk(int dev, int block) ->get_hash_table(dev,block) -> find_buffer(dev,block) -> hash(dev, block)*

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

*(P112)*

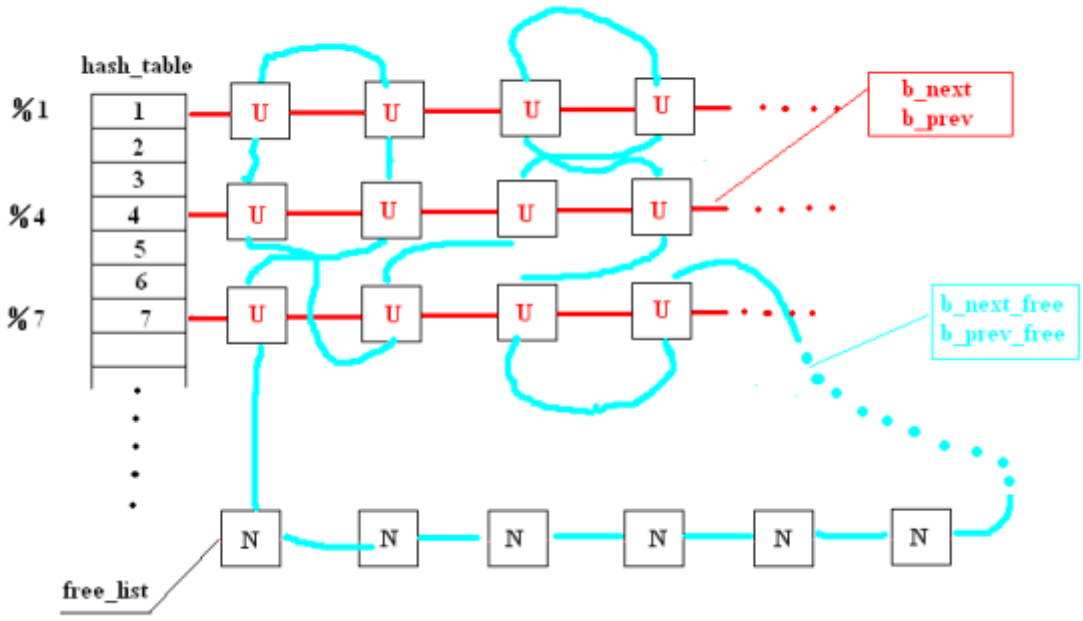



***B***

```
//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

}
```

总的效果图




***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*hash*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;

Bh->b_uptodate=0;

Remove_from_queues(bh);

Bh->b_dev=dev;

Bh->b_blocknr=block;

Insert_into_queues(bh);
```

**18、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';
```

**19、**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。

答：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;*

## 20、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。

## 21、add_request()函数中有下列代码

```
\linux0.11\kernel\blk_drv\ll_rw_blk.c

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？**(P322)**

> 答：查看指定设备是否有当前请求项，即查看设备是否忙。如果指定设备*dev*当前请求项（*dev->current_request ==NULL*） 为空，则表示目前设备没有请求项，本次是第*1*个请求项，也是唯一的一个。因此可将块设备当前请求指针直接指向该请求项，并立即执行相应设备的请求函数。

## 22、read_intr()函数中，下列代码是什么意思？为什么这样做？(P323)

```
\linux0.11\kernel\blk_drv\hd.c

if (--CURRENT->nr_sectors) {

          do_hd = &read_intr;

          return;

}
```

> 答：当读取扇区操作成功后，*“—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，于是再次置中断调用*C*函数指针*“do_hd = &read_intr;”*并直接返回，等待硬盘在读出另*1*扇区数据后发出中断并再次调用本函数。