操作系统思考题

- **1.1 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?**
  - 答：因为上电的时候，**内存 RAM** 中是空的，而 CPU 只能执行内存中的代码，不能执行硬盘或者软盘中的操作系统代码，所以就必须先执行 BIOS 代码，再通过 BIOS 代码加载操作系统代码。
  - 这就需要硬件主动加载 BIOS 程序，由 BIOS 准备好中断向量表和中断服务程序，接着**通过中断将引导程序 bootsect 加载到内存**里。
  - 再通过后续的一系列执行，操作系统的代码才能位于内存中，供 CPU 执行。
- **1.2 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?**
  - 两头约定，定位识别
  - 因为**操作系统和 BIOS 通常是由不同的团队**进行开发的。为了能够协调工作，双方按照固定的约定进行代码的开发。
  - 对于 BIOS 来说，它接收到启动命令后就**将启动扇区的代码加载至 0x07c00（BOOTSEG）处**，至于启动扇区里的内容是什么，BIOS 一概不管。
  - 而后续的代码则由**操作系统自己的 bootsect 代码进行加载**，这些代码由编写操作系统的团队负责。
  - 这样构建可以让 **BIOS 和操作系统**的设计团队**按照自己的意愿进行代码设计**，使**内存规划**更加灵活。
  - **为什么不全部加载?**
    - 对于**不同的操作系统**，其代码长度不一样，可能导致操作系统加载不完全。
    - 使用 BIOS 进行加载，且加载完成之后再执行，需要很长的时间，因此 Linux 采用的是边执行边加载的方法。
- **1.3 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?**
  - 答：0x07c00 是“两头约定” “定位识别”中要求的从启动扇区都要被加载到指定位置 0x07c00.
  - BIOS 在 **0x00000** 开始的位置构建起了**中断向量表**，暂时不能被覆盖，因此不能把 bootsect 加载到 0x00000 位置。
  - 只要接到启动操作系统的命令，BIOS 都要“被迫”做这样的工作，而 0x00000 是 BIOS 存放中断向量的地方
  - 加载后又马上挪到 0x90000 是操作系统开始根据自己的需要安排内存。
  - 不加载到位的原因是为了适配不同的操作系统，BIOS 并不知道加载的是哪个操作系统以及后续的内存规划，所以按照约定加载到 0x07c00 即可，后续由操作系统接管。
- **1.4 bootsect、setup、head 程序之间是怎么衔接的？给出代码证据。**
  - 答：bootsect -> setup : jmpi　　0,SETUPSEG
  - setup -> head: jmpi　 0,8
  - **bootsect→setup 程序：jmpi 0,SETUPSEG**
    - 其中 SETUPSEG 位置（setup 程序位置）：

而关于 SETUPSEG 的定义

```
1  //代码路径：boot\bootsect.s 34-39
2  SETUPLEN = 4                ! nr of setup-sectors [setup 程序的扇区数(setup-sectors)值]
3  BOOTSEG  = 0x07c0           ! original address of boot-sector[bootsect 的原始地址(是段地址，以下同)]
4  INITSEG  = 0x9000           ! we move boot here - out of the way [将 bootsect 移到这里 -- 避开]
5  SETUPSEG = 0x9020           ! setup starts here [setup 程序从这里开始]
6  SYSSEG   = 0x1000           ! system loaded at 0x10000 (65536).[system 模块加载到 0x10000(64 kB)处]
7  ENDSEG   = SYSSEG + SYSSIZE     ! where to stop loading [停止加载的段地址]
```

- **说明：**
- 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。

- **setup→head 程序：jmpi 0,8**
- 执行 setup 后，内核被移到了 0x00000 处，系统进入了保护模式，执行 jmpi 0,8。
- 在保护模式下，一个重要的特征就是根据 **GDT（全局描述符表）**决定后续执行哪里的程序。
- 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。

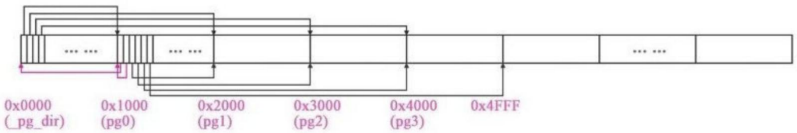
- **1.5. setup 程序的最后是 jmpi 0,8 ，为什么这个 8 不能简单的当作阿拉伯数字 8 看待，究竟有什么内涵?**
- jmpi 0,8 中 0 是段内偏移，8 是段选择子，8 的二进制表示为 1000，最后两位 00 表示内核特权级，与之相对的是用户特权级 11；倒数第三位 0 表示 GDT，1 表示 LDT；1 表示所对应的表（这里为 GDT）的下标 1 项（GDT 项号为 0、1、2）。通过这个来确定代码段的段基址和段限长等信息。查 GDT 表得到段基址为 0x00000000，所以 jmpi 0,8 就是跳转到段基址为 0x00000000 且偏移为 0 的地方，这里存放的是 head 程序初始地址，意味着从 head 程序开始执行。
- **1.6 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?**
- 保护操作系统的安全，不受到恶意攻击。**保护进程地址空间**。
- 体现在
- 打开了保护模式后，CPU 的寻址模式发生了变化，需要依赖于 **GDT 去获取代码或数据段的基址**。
- 从 GDT 可以看出，保护模式除了段基址外，还有段限长，这样相当于增加了一个段位寄存器。
- 既有效地防止了对代码或数据段的覆盖，又防止了代码段自身的访问超限，明显增强了保护作用。
- 在 GDT（全局描述表）、LDT（本地描述表）及 IDT（中断描述表）中，均有自己界限、特权级等属性，这是对描述符所描述的对象的保护；
- 在不同特权级间访问时，系统会对不同层级的程序进行保护，同还限制某些特殊指令的使用，如 lgdt, lidt,cli 等。

- **特权级的目的和意义是什么?**
- **目的是**为了进行**合理的管理资源，保护高特权级的段**。其中操作系统的内核处于最高的特权级。
- **意义：**
- cpu **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 等对掌控局面至关重要的指令。

- 有了这些基础，操作系统可以把**内核设计成最高特权级**，把**用户进程设计成最低特权级。**
- 这样，操作系统可以访问 GDT、 LDT、 TR，而 GDT、 LDT 是逻辑地址形成线性地址的关键，因此操作系统可以掌控线性地址。物理地址是由内核将线性地址转换而成的，所以操作系统可以访问任何物理地址。
- 而用户进程只能使用逻辑地址。总之，特权级的引入对操作系统内核进行保护。

- **分页有保护作用吗?**
  - 分页机制中 PDE（页目录索引）和 PTE（页表索引）中的 R/W（读写位）和 U/S（用户、特权级别位）等，提供了**页级保护 111**。
  - 分页机制将**线性地址与物理地址加以映射**，提供了对物理地址的保护。
- **为什么特权级是基于段的?（补充）**
  - 在操作系统设计中，一个段一般实现的功能相对完整，可以把代码放在一个段，数据放在一个段，并通过段选择符（包括 CS、SS、DS、ES、Fs 和 GS）**获取段的基址和特权级等信息。**
  - 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，这样当段选择具有不匹配的特权级时，按照特权级规则评判是否可以访问。
  - 特权级基于段，是结合了程序的特点和硬件实现的一种考虑。

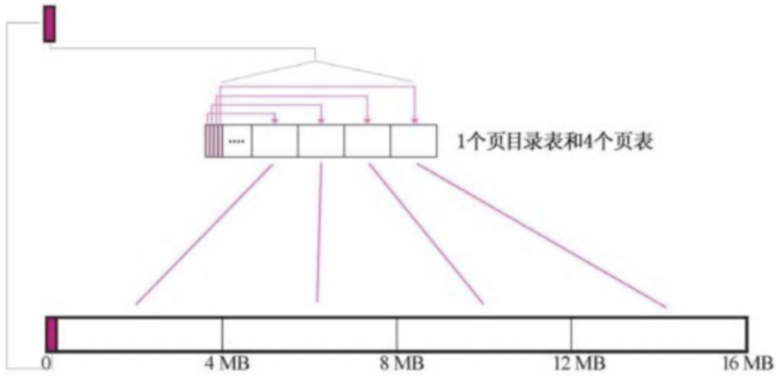
- **1.7 在 setup 程序里曾经设置过 gdt，为什么在 head 程序中将其废弃，又重新设置了一个？为什么设置两次，而不是一次搞好?**
  - 书本 P33
  - 废弃 setup 中设置的 GDT 的原因：原来 GDT 的位置是在 setup 模块里，将来 setup 模块的区域会在设计缓冲区的时候被覆盖，如果不改变 GDT 的位置，那么 GDT 也会被覆盖，从而影响系统运行。而整个内存中**唯一安全的地方就是 head 所在的位置**。
  - GDT 必须重新建立的原因：**GDT 和 system**
  - 如果在执行 setup 程序的时候直接将 GDT 一次搞好，那么移动 system 模块的时候会被覆盖；
  - 如果先移动 system 后复制 GDT，system 中的内容会被覆盖，但是此时 system 还没执行。
- **1.8 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。
  - **挂接关系图**



  - 代码证据：Head.s 中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（页表从 0 开始，即页表 0-页表 3）

```
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  /*  --------- " " --------- */
_pg_dir用于表示内核分页机制完成后的内核起始位置，也就是物理内存的起始位置0x000000，以上四句完成页目录表的前四项与页表1，2,3,4
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 */
```

- **1.9、根据内核分页为线性地址恒等映射的要求，推导出四个页表的映射公式，写出页表的设置代码。**
  - 内核分页采用线性地址恒等映射。
  - 内核的段基址是 0，代码段和数据段的段限长都是 16 MB。每个页面大小为 4 KB，每个页表可以管理 1024 个页面，每个页目录表可以管理 1024 个页表。既然确定了段限长是 16 MB，这样就需要 4 个页目录项（attention：只用了四个页目录项管理 4 个页表）下辖 4 个页表，来管理这 16 MB 的内存。



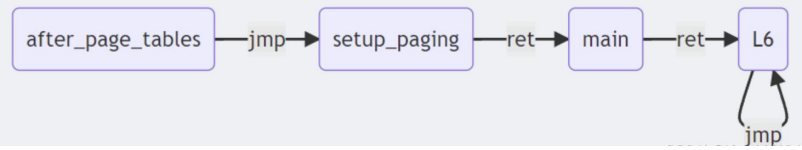
内核控制线性地址0～16MB的空间，也是物理地址的0～16MB的空间

  - 页表设置代码：内核分页采用恒等映射模式，调用 get_free_page( )函数后，获取的线性地址值直接就可以当物理地址来用

```
//代码路径: boot/head.s:
…
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
…
```

- **1.10、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，然后执行被调函数的程序，等到执行被调函数的 ret 指令时，自动出栈给 EIP 并还原现场，继续执行 call 的下一条指令。

- 然而对操作系统的 main 函数来说，如果用 call 调用 main 函数，那么 ret 时返回给谁呢？因为没有更底层的函数程序接收操作系统的返回。用 ret 实现的调用操作当然就不需要返回了，call 做的压栈和跳转动作需要手工编写代码。
- **调用路线图**

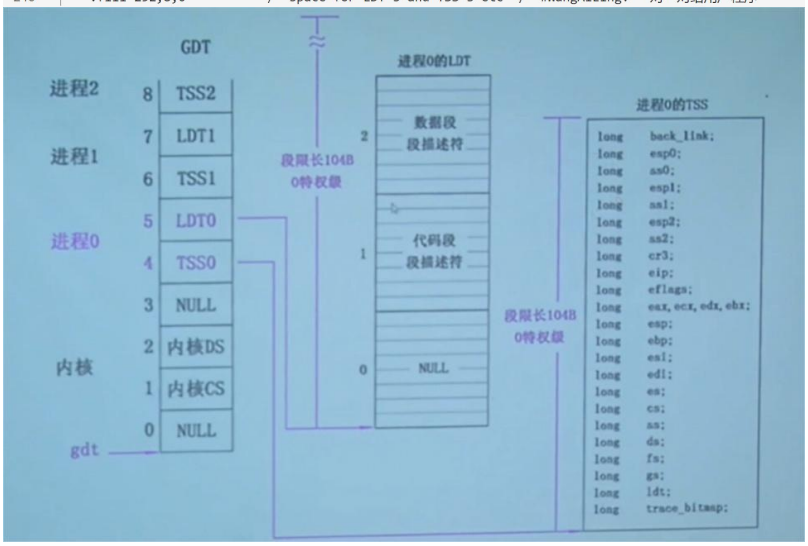


- **代码**

```
after_page_tables:
    pushl $0        # These are the parameters to main :-)  envp
    pushl $0        # argv
    pushl $0        # argc
    pushl $L6       # return address for main, if it decides to.
    pushl $_main    # kernel 的 main 函数地址
    jmp setup_paging
L6:
    jmp L6          # main should never return here, but
                # just in case, we know what happens.

setup_paging:   // 内核分页，分完以后 线性地址 == 物理地址
    // ...
    ret         /* this also flushes prefetch-queue */      // 我们是操作系统的底层，所以要返回到 kernel 中

```

- **2.1、计算内核代码段、数据段的段基址、段限长、特权级。**
  - 答案：代码段和数据段都是：Base = 0 Limit = 0ffff * 4k=16MB DPL=00
  - 首先找到内核代码段、数据段的定义位置　head.s 中：（至于为什么在 head.s 中，记得第一次思考题说到的为什么要在 head 中废弃 setup 设置的 gdt 然后重新设置吗 **编号 1.7**）

```
_gdt:   .quad 0x0000000000000000    /* NULL descriptor */
    .quad 0x00c09a0000000fff    /* 16Mb */
    .quad 0x00c0920000000fff    /* 16Mb */
    .quad 0x0000000000000000    /* TEMPORARY - don't use */
    .fill 252,8,0           /* space for LDT's and TSS's etc */  #WangAiLing: 一对一对给用户程序
```



  - 根据计算方法：

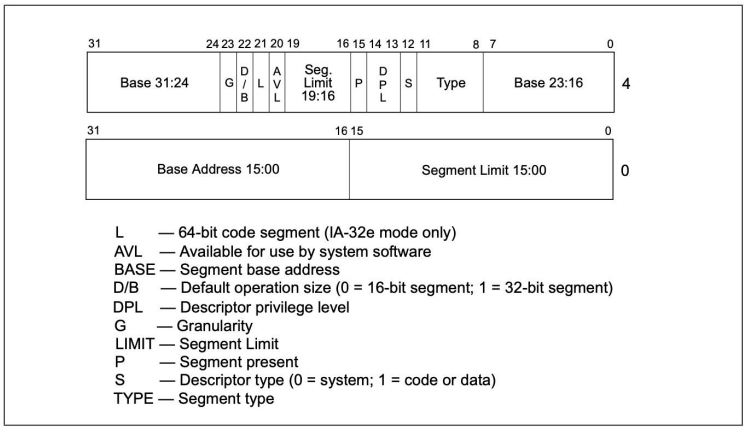


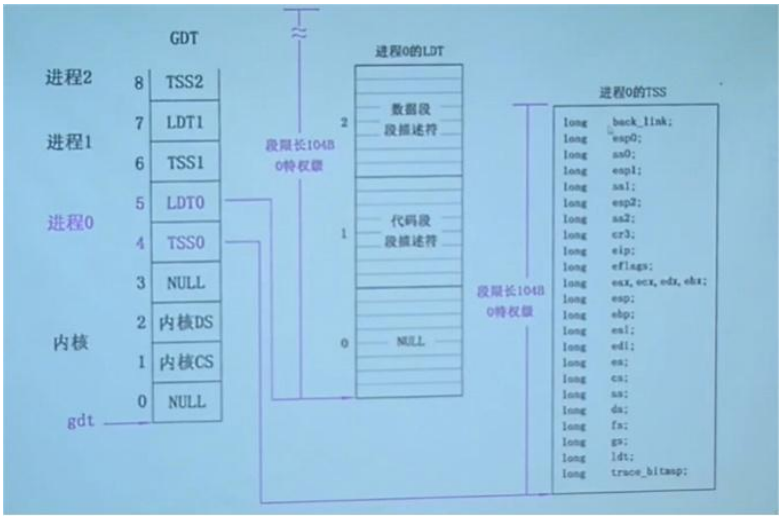
Figure 3-8. Segment Descriptor

- 算得（我懒得算了，在下一题有详细的计算方法，就是每一位拼接就可以了，展开得到结果）
  - Base: 0x00
  - Limit = Segment Limit : 0x0fff -> 2^12 * 4KB = 16MB
  - dpl：0

- **2.2、计算进程 0 的代码段、数据段的段基址、段限长、特权级。**
  - 答案：代码段和数据段都是 Base = 0 Limit = 0x9ffff = 640kB = 0x9f * 4k DPL=11=3
  - 首先找到位置，红色部分，CS 为代码段描述符 DS 为数据段描述符：

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

DS
CS
0 0



- 计算形式：

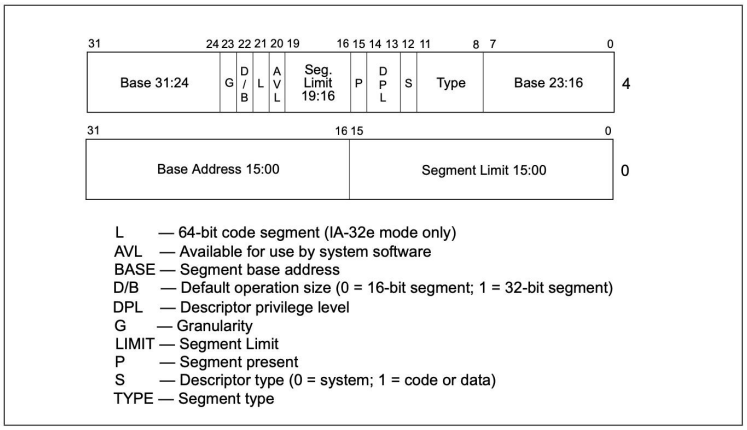


Figure 3-8. Segment Descriptor

- CS 代码段描述符 （注意是反着来的）00c0_fa00_0000_009f，只关注 Base 部分和 Segment Limit 部分：
  - Base：即二进制 00c0_fa00 的前八位和后八位以及 0000_009f 的前 16 位（从左往后看）
  - Segment Limit：0000_009f 的后十六位以及 16-19 位，组成了 0x09f = 160 * 4k = 640k
  - DPL：11
- DS 数据段描述符：同理得到：Base、Segement Limit、DPL 同代码段描述符

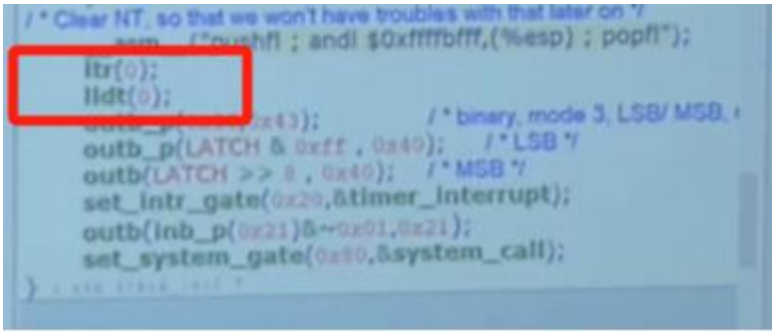
- **2.3、fork 进程 1 之前，为什么先调用 move_to_user_mode()？用的是什么方法？解释其中的道理。**
  - 因为 Linux 操作系统规定：除了进程 0，所有进程都要由一个 3 特权级的父进程创建
  - 要通过仅有的进程 0 fork 进程 1，那么进程 0 必须翻转到 3 特权级
  - 由于进程 0 的代码和数据都是由操作系统设计者写在内核代码和数据区的，并且在 sched_init()等操作后还处在 0 特权级，所以要通过调用 move_to_user_mode()翻转到 3 特权级，然后再创建进程 1
  - move_to_user_mode 使用的方法是**模拟中断压栈**，然后通过 **iret** 出栈和中断返回的操作来翻转特权级。
  - 中断返回的 cpu 硬件操作有：1. 硬件保护现场和恢复现场 2. 翻转特权级
  - move_to_user_mode 模拟中断压栈各个寄存器的值代码：按顺序压入 ss esp eflags cs eip

```
#define move_to_user_mode() \
__asm__ ("movl %%esp,%%eax\n\t" \
    "pushl $0x17\n\t" \
    "pushl %%eax\n\t" \
    "pushfl\n\t" \
    "pushl $0x0f\n\t" \
    "pushl $1f\n\t" \
    "iret\n" \
    "1:\tmovl $0x17,%%eax\n\t" \
    "movw %%ax,%%ds\n\t" \
    "movw %%ax,%%es\n\t" \
    "movw %%ax,%%fs\n\t" \
    "movw %%ax,%%gs" \
    :::"ax")
```

- **2.4、根据什么判定 move_to_user_mode()中 iret 之后的代码为进程 0 的代码。**
  - iret 之后的代码：光标之后

```
#define move_to_user_mode() \
__asm__ ("movl %%esp,%%eax\n\t" \
    "pushl $0x17\n\t" \
    "pushl %%eax\n\t" \
    "pushfl\n\t" \
    "pushl $0x0f\n\t" \
    "pushl $1f\n\t" \
    "iret\n" \
    "1:\tmovl $0x17,%%eax\n\t" \
    "movw %%ax,%%ds\n\t" \
    "movw %%ax,%%es\n\t" \
    "movw %%ax,%%fs\n\t" \
    "movw %%ax,%%gs" \
    :::"ax")
```

  - 三个证据（老师讲的）：
  - **证据 1（最关键）**：
    - 最关键的：**当前的 ldtr** 和 tr 都**是指向进程 0 的**，是在 sched_init 函数的后半部分设置的
    - 这两个寄存器指向的是当前进程的 LDT 和 TSS 段
    - ldtr tr 分别管 LDT 和 TSS



```
void sched_init(void)
{
    int i;
    struct desc_struct * p;

    if (sizeof(struct sigaction) != 16)
        panic("Struct sigaction MUST be 16 bytes");
    set_tss_desc(gdt+FIRST_TSS_ENTRY,&(init_task.task.tss));
    set_ldt_desc(gdt+FIRST_LDT_ENTRY,&(init_task.task.ldt));
    p = gdt+2+FIRST_TSS_ENTRY;
    for(i=1;i<NR_TASKS;i++) {
        task[i] = NULL;
        p->a=p->b=0;
        p++;
        p->a=p->b=0;
        p++;
    }
/* Clear NT, so that we won't have troubles with that later on */
    __asm__("pushfl ; andl $0xffffbfff,(%esp) ; popfl");
    ltr(0);
    lldt(0);
    outb_p(0x36,0x43);      /* binary, mode 3, LSB/MSB, ch 0 */
    outb_p(LATCH & 0xff , 0x40);    /* LSB */
    outb(LATCH >> 8 , 0x40);    /* MSB */
    set_intr_gate(0x20,&timer_interrupt);
    outb(inb_p(0x21)&~0x01,0x21);
    set_system_gate(0x80,&system_call);
}
```

- 证据 2：
  - 0x17 : 10111: 用户程序的数据段，此时只有进程 0，进程 1 未创建
- 证据 3：
  - INIT_TASK 中的 ldt

- **2.5、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 是操作系统设计者事先设计好的，位于**内核数据区。**
  - 进程 0 的 task_struct 的具体内容包含了进程 0 的**状态、信号、pid（进程号）、alarm（定时器）、filp、ldt（局部描述符表）、tss（任务状态段）**等管理该进程所需的数据。
  - 代码证据：sched.h

```
// 进程0的task_struct
#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, \
        {} \
    }, \
}
```

- **2.6、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。**
  - 代码：

```
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)
```

- dpl = 3：表示 3 特权级，对应用户级
- dpl = 0：表示 0 特权级，对应内核级
- 中断和异常处理是由内核来完成的，Linux 出于对内核的保护，不允许用户进程直接访问内核，只能内核使用，因此需要设置为 0 特权级。
- 但是在某些情况下，用户进程需要内核代码的支持。系统调用就是用户进程和内核打交道的接口，是允许用户进程调用的，因此为 3 特权级。

- **2.7、（重复）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。**
- **2.8、（重复）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。详细论证其他所有参数是如何传入的。**
- **2.9、详细分析 Linux 操作系统如何设置保护模式的中断机制。**
- **2.10、分析 Linux 操作系统如何剥夺用户进程访问内核及其他进程的能力。**
- **2.11、分析后面两行代码的意义。_system_call:　cmpl $nr_system_calls-1,%eax　ja bad_sys_call**
  - int80 之后进入系统调用 system_call 中的前两行代码，由于 system_call 是所有系统调用的总入口，同时所调用的编号是放在 eax 中的，所以需要先 cmpl 判断传入的这个编号是不是处于合法的系统调用的范围的，nr_system_calls 是系统调用的程序的数量 nr 指 number（一个确定性的固定的数值），如果 ja a 的意思是 above，即系统调用的范围超过了所有的能够调用的有效范围，就说明出错了，非法的访问，跳转到 bad_sys_call 中，组织非法调用；如果是正常的则正常执行系统调用的流程

```
_system_call:
    cmpl $nr_system_calls-1,%eax
    ja bad_sys_call
    push %ds
    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)
    pushl %eax
    movl _current,%eax
    cmpl $0,state(%eax)     # state
    jne reschedule
    cmpl $0,counter(%eax)       # counter
    je reschedule
```

- **3. 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。**
  - 这五个参数是 main 的 fork 函数中进行 int0x80 系统调用的时候压入栈的。int0x80 是一个中断，顺序压入：ss esp eflags cs eip 这些寄存器的值进内核栈中。同时因为函数的参数传递也是使用栈的，所以可以将这些寄存器的值作为 copy_process 的函数参数
- **3.2、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。**
  - 代码如下：

```
/*
 * Get physical address of first (actually last :-) free page, and mark it
 * used. If no free pages left, return 0.
 */
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;
}
```

位图最后一位从后向前找

如果为0没找到则跳转到1之后

  - 总的来说，获取空闲页的流程是：遍历 mem_map[]，从后向前（高地址空间）找到主内存中第一个空闲页面
  - 细节：
  - 首先，（第二个冒号）设置 eax 的值为 0，设置 edx 为 mem_map[]最后一个元素

- std; repne; scasb 反向扫描 mem_map[] 位图的引用计数不为 0 的时候重复
- 结束查找，此时找的结果有两种：
- 1. 没有找到空闲的物理页：eax 仍然为 0，那么就通过 jne 1f 跳转到到 1 之后，将 eax 的值 0 赋值给 res，并且 return __res 返回 0
- 2.找到了空闲的物理页：
  - movb $1,1(%%edi) 将找到的 0 的项的引用计数设置为 1
  - 12 ： 将 ecx 左移 12 位，得到页的相对地址
  - %2 第二个参数：将 LOW_MEM 加上 ecx，得到页的物理地址
  - rep;stosl 是将 eax 的值 0 赋值给 edi，目的是页面清零 ？从高到低反向页面清零?
  - movl %%edx,%%eax：将 edx 的值赋值给 eax
  - 然后 eax 的值赋值给 res，通过 return __res 返回

- **3.3、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。**
  - 代码如下：挨个分析即可

```
int copy_page_tables(unsigned long from,unsigned long to,long size)
{
    unsigned long * from_page_table;   父进程数据段基址   子进程数据段基址
    unsigned long * to_page_table;
    unsigned long this_page;
    unsigned long * from_dir, * to_dir;
    unsigned long nr;

    if ((from&0x3fffff) || (to&0x3fffff))   保证线性地址空间为4M的整数倍
        panic("copy_page_tables called with wrong alignment");
    from_dir = (unsigned long *) ((from>>20) & 0xffc); /* _pg_dir = 0 */   页目录表项的范围，最多是64M/4M=16
    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))   如果没分配
            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;   A0=160项
        for ( ; nr-- > 0 ; from_page_table++,to_page_table++) {
            this_page = *from_page_table;
            if (!(1 & this_page))
                continue;
            this_page &= ~2;   ~2=101用户只读存在
            *to_page_table = this_page;
            if (this_page > LOW_MEM) {   LOW_MEM是1MB，1MB以下不参加分页管理
                *from_page_table = this_page;
                this_page -= LOW_MEM;
                this_page >>= 12;
                mem_map[this_page]++;   引用计数+1
            }
        }                                   页面共享
    }
    invalidate();
    return 0;   刷新CR3 线性地址 物理地址对应 页变换高速缓存
}
```

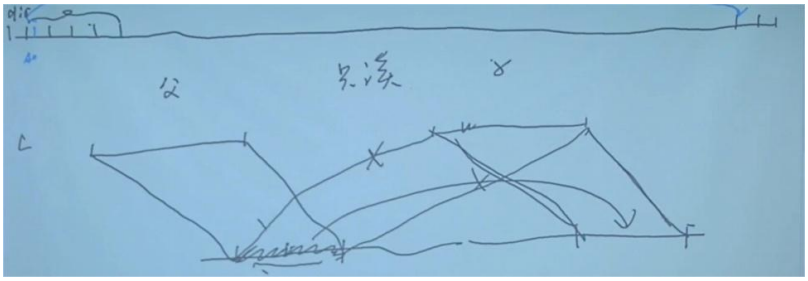
父子进程一一对应

父进程的某个页目录表项的值：页表

为子进程分配一个物理页：用作页表

页表属性：111用户读写存在

  - 第一个 if 还没给子进程的页表分配就已经有了页 P 位为 1，panic
  - 第二个 if 如果父进程的页表是空的，那就不需要给子进程分配物理页，跳过
  - 进程 0 也就 640K，一个页都够了，这里是会循环 16 个页目录表项，16*4M=64M 的线性地址空间?
  - 7:111User 用户 可读写 存在
  -
  - 设置为只读的原因是：
  - 为了不让子进程进行改写从而影响父进程
  - 约束线性地址，因为一个线性地址对应一个物理地址，不存在一个线性地址对应多个物理地址，但是一个物理地址可以对应多个线性地址，多个进程，约束了线性地址就可以访问控制了



- 32 位实际上 20 位就够了，
- 写实复制 COW，如果谁要写，就直接 get free page 脱钩新开一个页

- **3.4、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，分别位于物理内存两个页。请问，这两个页的位置，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。**
  - 内核的线性地址空间
  - 在 setup_pages 中对内核的 16M 的线性地址空间恒等映射物理空间
  - get_free_page 获取了一个物理页，返回的是内核的线性地址
  - 内核可以通过线性地址恒等找到物理页，但是对于 task_struct 和页表这两个页是只由内核管理的，并且后续没有任何将这个物理页映射到进程线性地址空间的内容。所以是仅占用内核的线性地址空间
- **3.5、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。**
- **6、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。**
  - 先看代码

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

  - 这个指令%0 的意思是第一个操作数， “::” 后面是操作数列表，所以 ljmp 会跳到第一个操作数 "m" (*&__tmp.a)
  - 这是 ljmp 的特殊用法，一般 ljmp 的参数是段选择子+偏移符。他跳转的其实是某个段选择子下的代码块。而如果涉及到进程切换的话，这个偏移符会被忽略，目标进程的 TSS 中的寄存器会被恢复，程序实际上会跳转到那个进程的 TSS 中的 eip 中。

- 这个长跳转执行的切换效果就是将处在内核态的进程 A 直接切换到用户态的进程 B 的代码中进行执行。

- **7、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。**
  - 第一次执行 fork 的时候

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

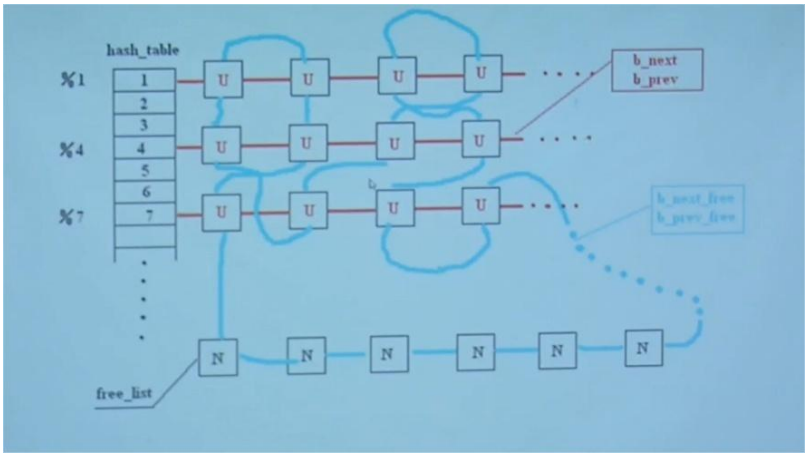
- **3.8、详细分析进程调度的全过程。考虑所有可能（signal、alarm 除外）**
  - 找到代码位置：

```
void schedule(void)
{
    int i,next,c;
    struct task_struct ** p;

/* check alarm, wake up any interruptible tasks that have got a signal */

    for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
        if (*p) {
            if ((*p)->alarm && (*p)->alarm < jiffies) {
                    (*p)->signal |= (1<<(SIGALRM-1));
                    (*p)->alarm = 0;
                }
            if (((*p)->signal & ~(_BLOCKABLE & (*p)->blocked)) &&
            (*p)->state==TASK_INTERRUPTIBLE)
                (*p)->state=TASK_RUNNING;
        }

/* this is the scheduler proper: */

    while (1) {
        c = -1;
        next = 0;
        i = NR_TASKS;
        p = &task[NR_TASKS];
        while (--i) {
            if (!*--p)
                continue;
            if ((*p)->state == TASK_RUNNING && (*p)->counter > c)
                c = (*p)->counter, next = i;
        }
        if (c) break;
        for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
            if (*p)
                (*p)->counter = ((*p)->counter >> 1) +
                        (*p)->priority;
    }
    switch_to(next);
}
```

信号部分不管

  - 第一种可能：队列中有就绪进程，且时间片不为 0。
    - 例如从进程 0 调度到进程 1，此时进程 0 挂起，进程 1 就绪。

        - i=NR_TASKS 的意思是 NUMBER_TASKS 进程号，p 是进程号对应的 task_struct 结构。
        - next 表示找到的下一个进程调度的进程号，c 表示找到的进程调度的进程的 counter 时间片。
        - （挑选目标：找出所有就绪进程中时间片最长的）
        - 挑选调度进程的步骤：从 63 号进程一直减，挨个看是不是 TASK_RUNNING 就绪态，是就绪态的话时间片是不是最长的
        - 在此种可能下，找到了就绪的进程 1，c 的值为进程 1 的时间片，不为 0，则 break
        - 然后 switch_to 调度到进程 1
    - 第二种可能：队列中有就绪进程，但是时间片为 0. 所有就绪进程的时间片都用完了
        - 此时 c 的值=0，进入 for 循环按照优先级为进程重新分配时间片
        - 然后循环，重新查看是否是第一种可能等
    - 第三种可能：队列中没有就绪进程
        - 例如进程 0 挂起，进程 1 挂起，且进程对列里没有其他进程就绪了。
        - 挑选完所有的进程后，发现没有找到任何的就绪态的进程，c 的值不变，仍为-1，-1 为真，则 break
        - 然后在进程 0 不就绪的情况下，直接 switch_to 调度到进程 0，后续如果没有就绪会进入怠速循环
- **3.9、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?**
    - 将 for 循环改为跳转到 内核 0 特权级进程 中执行
-
- **4.1、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?**
    - getblk：如果有，返回现成的，没有现成的缓冲块，返回空；返回空之后找没人用过的 free list，引用计数为 0 的，引用计数为 0 不见得设备号块号为 0，选 BADNESS 更优的，dirt 时间更长，lock 其次



    - 因为虽然引用计数为 0，但是可能这个块可能是被锁住的，锁住后就要 sleep on 等待这个 buffer，
- **4.2、b_dirt 已经被置为 1 的缓冲块，同步前能够被进程继续读、写吗？给出代码证据。**
    - b dirt 的意思是该块是否为脏，如果没有和设备块同步，则为脏 1；同步了则为 0
    - 只要 uptodate 为 1：buffer 已经更新，同步前就能够被进程继续读写
    - 代码证据：bread file_write file_read

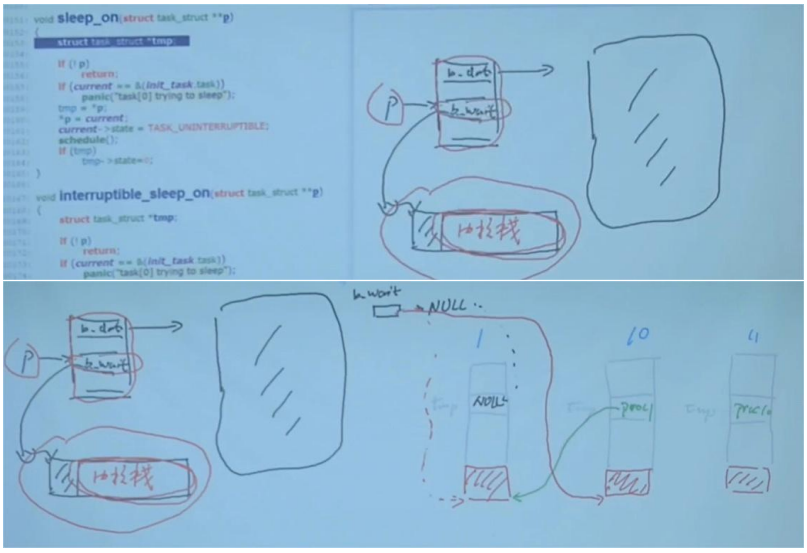
- **4.3、wait_on_buffer 函数中为什么不用 if（）而是用 while（）？**
  - 首先明确这个 wait_on_buffer 出现的位置：在 ll_rw_block 进入了读写操作后，硬件在一直读写，这边软件就执行下去等硬件，因为在 make_request 的时候 bh->b_block 已经被 lock 了为真

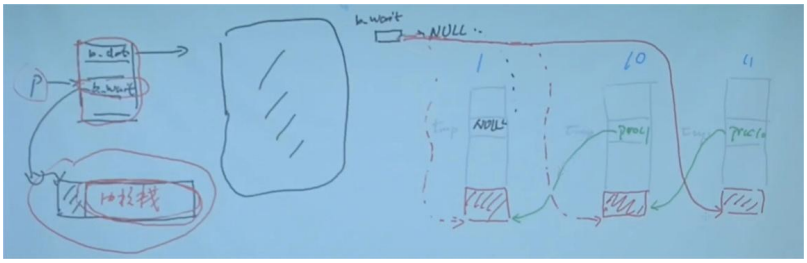
```
static inline void wait_on_buffer(struct buffer_head * bh)
{
    cli();
    while (bh->b_lock)
        sleep_on(&bh->b_wait);
    sti();
}
```

  - 进入 sleep on 之后会进行进程调度

```
void sleep_on(struct task_struct **p)
{
    struct task_struct *tmp;

    if (!p)
        return;
    if (current == &(init_task.task))
        panic("task[0] trying to sleep");
    tmp = *p;
    *p = current;
    current->state = TASK_UNINTERRUPTIBLE;
    schedule();
    if (tmp)
        tmp->state=0;
}
```

  - 那什么时候接着会走下去？等到执行这个 wait_on_buffer 的进程通过别的进程被调度
  - 但是由于这个 buffer 块不一定解锁了，可能当前进程解锁了，但是别的进程可能会加锁 (bwait 是一个
  - task struct 指针，其背后是一个等待对列，即当前 buffer 块要等待的不只一个进程)
  - 如果别的进程加锁了就需要一直等着
    - buffer-head 是全局的，bwait 进程等待队列也是全局的
    - tmp 局部变量在进程 1 的内核栈里面，很巧妙地通过 tmp 形成了等待对列

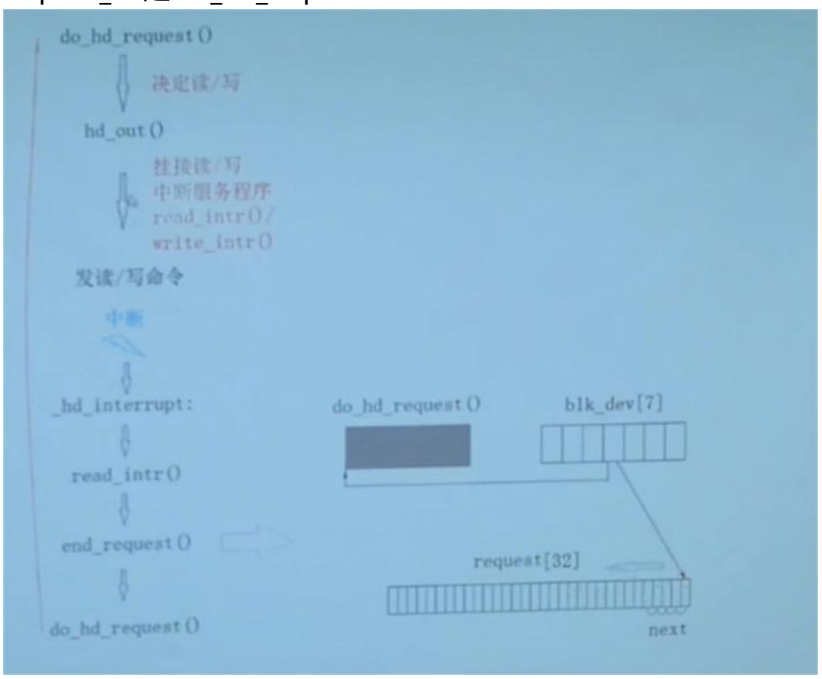


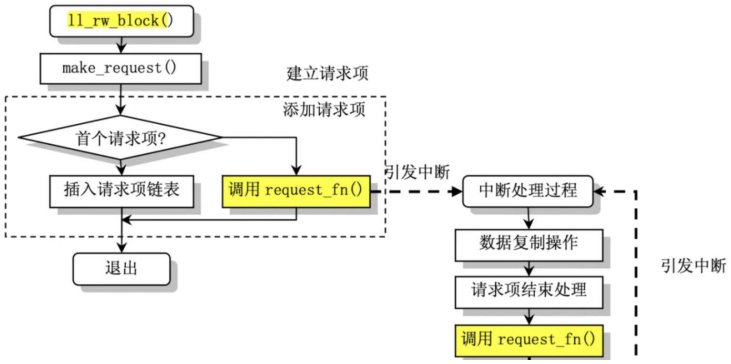


- **4.4、分析 ll_rw_block(READ,bh)读硬盘块数据到缓冲区的整个流程（包括借助中断形成的类递归），叙述这些代码实现的功能。10-30**
  - rw 的参数为 READ

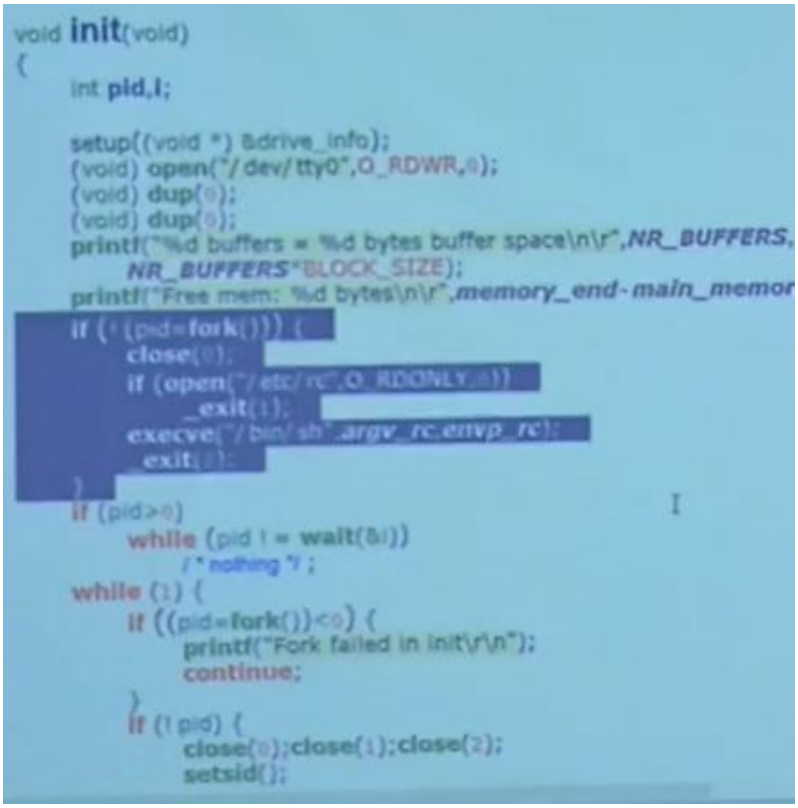
```
void ll_rw_block(int rw, struct buffer_head * bh)
{
    //底层的块设备的读写操作：体现在两部分：一个请求项，一个驱动
    unsigned int major;

    if ((major=MAJOR(bh->b_dev)) >= NR_BLK_DEV ||
    !(blk_dev[major].request_fn)) {
        printk("Trying to read nonexistent block-device\n\r");
        return;
    }
    // 重点关注请求项
    // 两部分：制作请求项和添加请求项
    make_request(major,rw,bh);
}
```

  - 进入 make_request 请求项部分,make_request 函数里面分为两部分:制作请求项和添加请求项
  - request_fn 是 do_hd_request





- **4.5、分析包括安装根文件系统、安装文件系统、打开文件、读文件在内的文件操作。**
  - **安装根文件系统**
    - 复制超级块到 super_block[8]中，将根设备等 inode 挂载到 superklock 对应的超级块上。
    - 将驻留缓冲区的 16 个缓冲块的根设备逻辑位图和 inode 位图挂载到 s_zmap[8]和 s_imap[8]中
    - 将当前进程的 pwd 和 root 修改到根 inode 中
  - **安装文件系统**
    - 超级块读出，装入 sb8 中。
    - 将虚拟盘上指定的 inode 读出，并加载到 inode_table[32]中。
    - 将硬盘上的超级块挂载到 inode table[32]中。
  - **打开文件**
    - taskstruct 中的 filp[20]与内核中的 file table[64]挂接。
    - 用户进程需要打开的文件对应的 inode 在 file table[64]中登记。
  - **读文件**
    - 由 read 实现，通过 bmp 找到文件在外设中对应的位置，然后通过 bread 读入缓冲区，最后复制到用户空间。
  - 安装根文件系统
  - 安装文件系统
  - 打开文件
    - 顺着 open 的需求通过 sys_open
    - taskstruct 中的 filp[20]与内核中的 file table[64]挂接。
    - 用户进程需要打开的文件对应的 inode 在 file table[64]中登记。
  - 读文件
    - 由 read 实现，通过 bmp 找到文件在外设中对应的位置，然后通过 bread 读入缓冲区，最后复制到用户空间。
- **4.6、在创建进程、从硬盘加载程序、执行这个程序的过程中，sys_fork、do_execve、do_no_page 分别起了什么作用?**
  -
  - do_execve 加载可执行文件 do_execve 用于加载并执行新的程序。在 init



```
void init(void)
{
    int pid,i;

    setup((void *) &drive_info);
    (void) open("/dev/tty0",O_RDWR,0);
    (void) dup(0);
    (void) dup(0);
    printf("%d buffers = %d bytes buffer space\n\r",NR_BUFFERS,
        NR_BUFFERS*BLOCK_SIZE);
    printf("Free mem: %d bytes\n\r",memory_end-main_memor
    if (!(pid=fork())) {
        close(0);
        if (open("/etc/rc",O_RDONLY,0))
            _exit(1);
        execve("/bin/sh",argv_rc,envp_rc);
        exit(2);
    }
    if (pid>0)
        while (pid != wait(&i))
            /* nothing */;
    while (1) {
        if ((pid=fork())<0) {
            printf("Fork failed in init\r\n");
            continue;
        }
        if (!pid) {
            close(0);close(1);close(2);
            setsid();
```

- do_no_page 用于处理程序执行过程中的缺页异常，确保所需的内存页面被正确加载。
-
-
-

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

- up to date 保持硬盘和缓冲块一致 在 end_request 的时候使用
-
-
- **fork 执行路线：**
  - fork 函数定义：将宏定义展开即为 C 函数 int fork(void) { long **res; ....return -1} C 函数中的内联汇编的第一个冒号后面是输出部分：将 int0x80 系统调用的返回值（eax 寄存器中的）赋值给 res；第二个冒号后面是输入部分：int0x80 的传入参数，传入__NR_fork 也就是 2，即 eax 寄存器的值为 2

```
static inline _syscall0(int,fork)
#define _syscall0(type,name) \
type name(void) \
{ \
long __res; \
__asm__ volatile ("int $0x80" \
    : "=a" (__res) \
    : "0" ( __NR ##name)); \
if (__res >= 0) \
    return (type) __res; \
errno = -__res; \
return -1; \
}
```

```
#define __NR_setup  0   /* used only by init, to get system going */
#define __NR_exit   1
#define __NR_fork   2
#define __NR_read   3
#define __NR_write  4
#define __NR_open   5
#define __NR_close  6
```

- **内核栈：ss esp eflags cs eip**
- 一旦执行 0x80，就从 3 特权级到了 0 特权级（int80 是在哪里设置的呢？sched init 函数最后一行，记得之前的题目，三个门然后 DPL 的题目吗？就是那里）int80 是所有系统调用的入口，执行的内容如下：

```
_system_call:
    cmpl $nr_system_calls-1,%eax
    ja bad_sys_call
    push %ds
    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)
    pushl %eax
    movl _current,%eax
    cmpl $0,state(%eax)     # state
    jne reschedule
    cmpl $0,counter(%eax)       # counter
    je reschedule
```

eax = 2

call _sys_call_table+2*4

= call _sys_fork

- **栈：ss esp eflags cs eip ds es fs edx ecx ebx none (call sys_fork)**

- 进入 int80 后找到合法的系统调用程序，这里是 **sys_fork**（sys_call_table 直接理解函数数组，里面有很多系统调用函数，要做的就是给数组的下标）sys_fork 做两件事：1. 申请 task[64]中的一个位置并分配一个可用的进程号 2.复制拷贝进程

```
_sys_fork:
    call  find_empty_process
    testl %eax,%eax
    js 1f
    push %gs
    pushl %esi
    pushl %edi
    pushl %ebp
    pushl %eax
    call _copy_process
    addl $20,%esp
1:  ret
```

  - 如果 last_pid+1 溢出 则赋值为 1，否则使用 last pid 增加 1 之后的值，这是进程号，然后验证进程号是否合法，如果有某个进程用了这个进程号就一直增加，直到找到合法的进程号；找到进程号后，在 task[64]中找一个空闲的位置（最多同时 64 个进程）如果没有，就返回一个负值

```
int find_empty_process(void)
{
    int i;

    repeat:
        if ((++last_pid)<0) last_pid=1;
        for(i=0 ; i<NR_TASKS ; i++)
            if (task[i] && task[i]->pid == last_pid) goto repeat;
    for(i=1 ; i<NR_TASKS ; i++)
        if (!task[i])
            return i;
    return -EAGAIN;
}
```

- **栈：ss esp eflags cs eip ds es fs edx ecx ebx none (call sys_fork) gs esi edi ebp eax(这里 eax 就是 find_empty_process 中找到的进程号)**
- 下面进行进程拷贝：
  - 参数的问题已经搞明白了，就是把栈里的内容统统使用而已：熟读这段话：最后五个参数是从 fork 的 int 0x80 带入的，倒数 2、3 行 6 个参数是刚进入 system_call 的时候压入的，long none 是 call sys_call_table 的时候压入的，没有返回，然后 find_empty_process 这个是返回了的，不需要体现，然后就是结束 find_empty_process 之后，copy process 之前 push 进入的 5 个参数，然后需要注意的是第一个参数 nr 实际上是 eax，也就是 find empty process 的返回值

```
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)
{
```

- 首先 get_free_page，这里如果是进程 0 创建进程 1，那么是首次调用 get_free_page，这边的解释在前面的题目中已经出现，这边的返回值是页的物理地址，然后直接将进程 0 的 task_struct 的 1k 赋值到进程 1 的 task_struct 中，复制完后进行个性化设置

```
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;
}
    task[nr] = p;
    *p = *current;  /* NOTE! this doesn't copy the supervisor stack */
```

  - 个性化设置需要注意：eip 的值为进程 0 的 eip，eax 为 0，进程状态开始为 TASK_UNINTERRUPTABLE

```
    p->state = TASK_UNINTERRUPTIBLE;
    p->pid = last_pid;
    p->father = current->pid;
    p->counter = p->priority;
    p->signal = 0;
    p->alarm = 0;
    p->leader = 0;      /* process leadership doesn't inherit */
    p->utime = p->stime = 0;
    p->cutime = p->cstime = 0;
    p->start_time = jiffies;
    p->tss.back_link = 0;
    p->tss.esp0 = PAGE_SIZE + (long) p;
    p->tss.ss0 = 0x10;
    p->tss.eip = eip;
    p->tss.eflags = eflags;
    p->tss.eax = 0;
    p->tss.ecx = ecx;
    p->tss.edx = edx;
    p->tss.ebx = ebx;
    p->tss.esp = esp;
    p->tss.ebp = ebp;
    p->tss.esi = esi;
    p->tss.edi = edi;
    p->tss.es = es & 0xffff;
    p->tss.cs = cs & 0xffff;
    p->tss.ss = ss & 0xffff;
    p->tss.ds = ds & 0xffff;
    p->tss.fs = fs & 0xffff;
    p->tss.gs = gs & 0xffff;
    p->tss.ldt = _LDT(nr);
    p->tss.trace_bitmap = 0x80000000;
    if (last_task_used_math == current)
        __asm__("clts ; fnsave %0"::"m" (p->tss.i387));
```

- 其次 **copy_mem**: 设置子进程的代码段、数据段以及创建和复制子进程的第一个页表

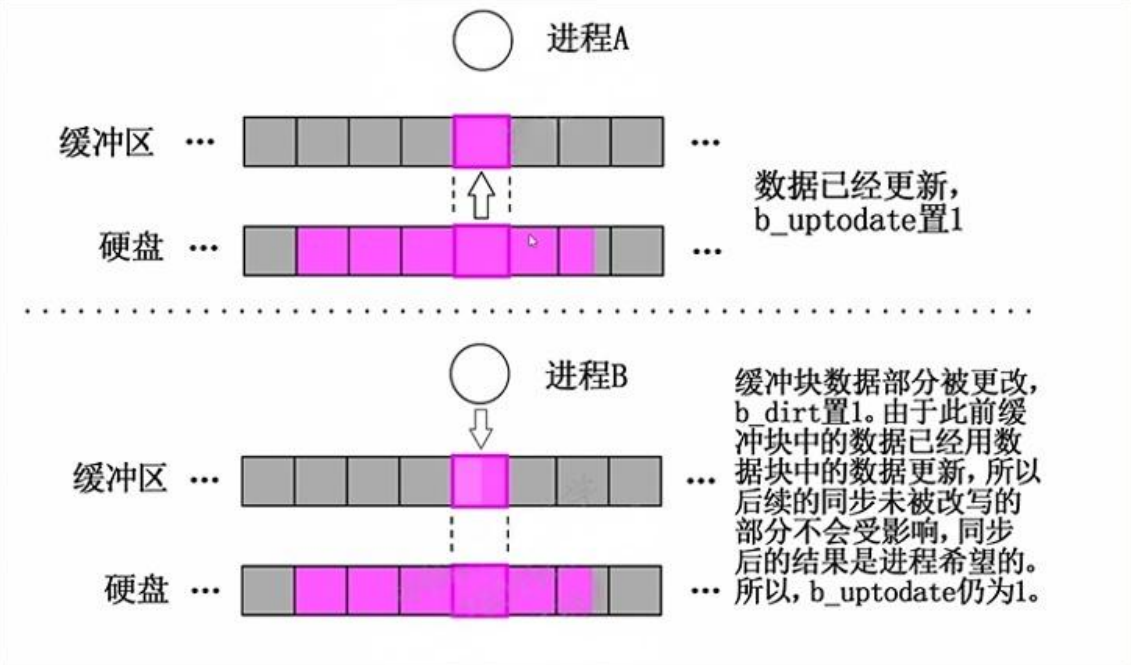
```
if (copy_mem(nr,p)) {
    task[nr] = NULL;
    free_page((long) p);
    return -EAGAIN;
}
int copy_mem(int nr,struct task_struct * p)
{
    unsigned long old_data_base,new_data_base,data_limit;
    unsigned long old_code_base,new_code_base,code_limit;

    code_limit=get_limit(0x0f);
    data_limit=get_limit(0x17);
    old_code_base = get_base(current->ldt[1]);
    old_data_base = get_base(current->ldt[2]);
    if (old_data_base != old_code_base)
        panic("We don't support separate I&D");
    if (data_limit < code_limit)
        panic("Bad data_limit");
    new_data_base = new_code_base = nr * 0x4000000;
    p->start_code = new_code_base;
    set_base(p->ldt[1],new_code_base);
    set_base(p->ldt[2],new_data_base);
    if (copy_page_tables(old_data_base,new_data_base,data_limit)) {
        free_page_tables(new_data_base,data_limit);
        return -ENOMEM;
    }
    return 0;
}
```

限长 f=(01 CS)(1 LDT)(11)

64M

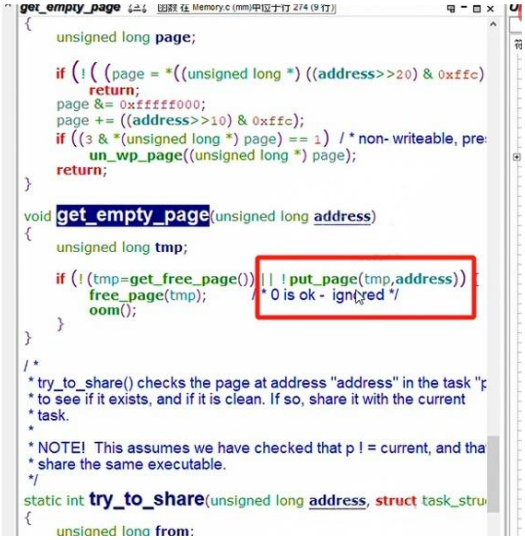
- 然后 filp 就是打开的文件的，一次最多 20
- 最后再挂上 LDT TSS，然后变成 RUNNING 就绪态

- bread 函数的作用：将硬盘中的内容读到缓冲区，uptodate 置 1



- put_page 的作用：将一个页映射到线性地址空间

```
00365: void do_no_page(unsigned long error_code,unsigned long address)
00366: {
00367:     int nr[4];
00368:     unsigned long tmp;
00369:     unsigned long page;
00370:     int block,i;
00371:
00372:     address &= 0xfffff000;
00373:     tmp = address - current->start_code;
00374:     if (!current->executable || tmp >= current->end_data) {
00375:         get_empty_page(address);
00376:         return;
00377:     }
00378:     if (share_page(tmp))
00379:         return;
00380:     if (!(page = get_free_page()))
00381:         oom();
00382: /* remember that 1 block is used for header */
00383:     block = 1 + tmp/BLOCK_SIZE;
00384:     for (i=0 ; i<4 ; block++,i++)
00385:         nr[i] = bmap(current->executable,block);
00386:     bread_page(page,current->executable->i_dev,nr);
00387:     i = tmp + 4096 - current->end_data;
00388:     tmp = page + 4096;
00389:     while (i-- > 0) {
00390:         tmp--;
00391:         *(char *)tmp = 0;
00392:     }
```



```
get_empty_page
{
    unsigned long page;

    if (!( (page = *((unsigned long *) ((address>>20) & 0xffc)
        return;
    page &= 0xfffff000;
    page += ((address>>10) & 0xffc);
    if ((3 & *(unsigned long *) page) == 1)  /* non- writeable, pre
        un_wp_page((unsigned long *) page);
    return;
}

void get_empty_page(unsigned long address)
{
    unsigned long tmp;

    if (!(tmp=get_free_page()) || !put_page(tmp,address)) {
        free_page(tmp);         /* 0 is ok - ignored */
        oom();
    }
}

/*
 * try_to_share() checks the page at address "address" in the task "p
 * to see if it exists, and if it is clean. If so, share it with the current
 * task.
 *
 * NOTE! This assumes we have checked that p != current, and tha
 * share the same executable.
 */
static int try_to_share(unsigned long address, struct task_stru
{
    unsigned long from;
```

Surprise！

恭喜你看到最后！真题大放送！闭卷！24

1. jmpi 0,8 (10)　你懂的

2. copy_process 的最后五个参数 (10)　你懂的

3. 为什么 fork()反复执行 (20) 无代码，需要自己写，最好针对性记一下　你懂的

4. 解释 copy_page_tables 的代码功能 有代码 (20)　你懂的

5. 三个进程等待同一个缓冲区，如何形成、唤醒等待队列？给了 wait_on_buffer 和 sleep_on 代码，要求画图解释 （老师上课讲的 cai 徐坤） (20)　似懂非懂

6. 解释以下代码在加载进程代码中的作用 给了 do_no_page　get_empty_page put_page bread_page (20)　真不懂了