1为什么计算机启动最开始的时候执行的是BIOS代码而不是操作系统自身的代码？

计算机启动的时候，内存未初始化， CPU不能直接从外设运行操作系统，所以必须将操作系统加载至内存中。而这个工作最开始的部分，BIOS需要完成一些检测工作，和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值 0x7C00 处，然后将跳转至 0x7C00。这些就是由bios程序来实现的。所以计算机启动最开始执行的是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没有把所有需要加载的扇区都加载？

对BIOS而言，“约定”在接到启动操作系统的命令后，“定位识别”只从启动扇区把代码加载到0x7c00这个位置。后续扇区则由bootsect代码加载，这些代码由编写系统的用户负责，与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。BIOS和操作系统的开发都可以遵循这一约定，灵活地进行各自的设计。操作系统的开发也可以按照自己的意愿，内存的规划，等等都更为灵活

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？

1）因为BIOS将从0x00000开始的1KB字节构建了了中断向量表，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。

2）首先，在启动扇区中有一些数据，将会被内核利用到。

其次，依据系统对内存的规划，内核终会占用0x0000其实的空间，因此0x7c00可能会被覆盖。

将该扇区挪到0x90000，在setup.s中，获取一些硬件数据保存在0x90000~0x901ff处，可以对一些后面内核将要利用的数据，集中保存和管理。

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;

这条语句跳转到0x90200处，即setup程序加载的位子，CS:IP指向setup程序的第一条指令，意味着setup开始执行。

2)setup跳转到head程序：CPU工作模式首先转变为保护模式然后执行 jmpi 0,8

0指的是段内偏移，8是保护模式下的段选择符：01000，其中后两位表示内核特权级，第三位0代表GDT，1则表示GDT表中的第一项，即内核代码段，段基质为0x0000000,而head程序地址就在这里，意味着head程序开始执行。

5.setup程序里的cli是为了什么？

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。

6.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？

这里8要看成二进制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表示所选的表（在此就是GDT表）的1项来确定代码段的段基址和段限长等信息。这样，我们可以得到代码是从段基址0x00000000、偏移为0处开始执行的，即head的开始位置。注意到已经开启了保护模式的机制，所以这里的8不能简单的当成阿拉伯数字8来看待。

7.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？

1、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。打开PE是进入保护模式。A20是cpu的第21位地址线，A20未打开的时候，实模式中cs：ip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。当打开A20的时候，实模式下cpu可以寻址到1MB以上的高端内存区。A20未打开时，如果打开pe，则cpu进入保护模式，但是可以访问的内存只能是奇数1M段，即0-1M,2M-3M,4-5M等。A20被打开后，如果打开pe，则可以访问的内存是连续的。打开A20是打开PE的必要条件；而打开A20不一定非得打开PE。

2、有必要。打开PE只是说明系统处于保护模式下，但若真正在保护模式下工作，必须打开A20，实现32位寻址。 

8.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，计算机处在了32位的保护模式状态了，调用main的条件就算准备完毕。

9.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。（图在P42）

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。

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。

10.保护模式的“保护”体现在哪里？

1）在 GDT、 LDT 及 IDT 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

2）在不同特权级间访问时，系统会对 CPL、 RPL、 DPL、 IOPL 等进行检验，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

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。

P438

11.特权级的目的和意义是什么？为什么特权级是基于段的？

目的：在于保护高特权级的段，其中操作系统的内核处于最高的特权级。

意义：保护模式中的特权级，对操作系统的“主奴机制”影响深远。

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。特权级基于段，这样当段选择子具有不匹 配的特权级时，按照特权级规则评判是否可以访问。特权级基于段，是结合了程序的特点和硬件 实现的一种考虑。

12.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？

原来的 GDT 位于 setup 中，将来此段内存会被缓冲区覆盖， 所以必须将 GDT 设置 head.s 所在 位置。

如果先将 GDT 设置在 head 所在区域，然后移动 system 模块，则 GDT 会被覆盖掉，如果先移 动 system 再复制 GDT，则 head.s 对应的程序会被覆盖掉，所以必须重建 GDT。

若先移动 system 至 0x0000 再将 GDT 复制到 0x5cb8~0x64b8 处，虽可以实现， 但由于 setup.s 与 head.s 连接时不在同一文件， setup 无法直接获取 head 中的 gdt 的偏移量，需事先写入， 这会使设计失去一般性，给程序编写带来很大不便。

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

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，即task结构的第0项init\_task。

struct task\_struct \* task[NR\_TASKS] = {&(init\_task.task), };

具体内容：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。具体值如下：（课本P68）

15.进程0创建进程1时，为进程1建立了自己的task\_struct、内核栈，第一个页表，分别位于物理内存16MB的顶端倒数第一页、第二页。请问，这个了页究竟占用的是谁的线性地址空间，内核、进程0、进程1、还是没有占用任何线性地址空间（直接从物理地址分配）？说明理由并给出代码证据。

这两个页占用的是内核的线性地址空间，依据在setup\_paging（文件head.s）中，

movl $pg3+4092,%edi

movl $0xfff007,%eax /\* 16Mb -4096 + 7 (r/w user,p) \*/

std

1: stosl/\* fill pages backwards -more efficient :-) \*/

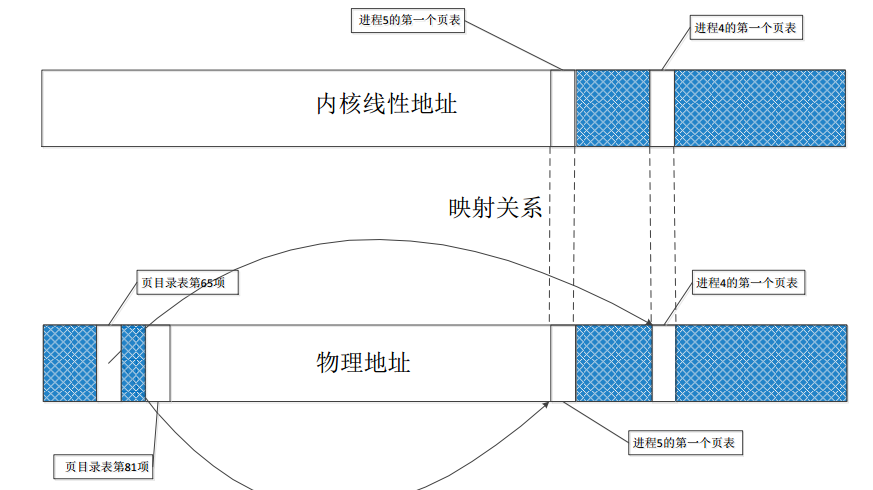
subl $0x1000,%eax

上面的代码，指明了内核的线性地址空间为0x000000 ~ 0xffffff（即前16M），且线性地址与物理地址呈现一一对应的关系。为进程1分配的这两个页，在16MB的顶端倒数第一页、第二页，因此占用内核的线性地址空间。

进程0的线性地址空间是内存前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（160项），而这160个页表项即为内核第一个页表的前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。

16.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空间。



17.进程0开始创建进程1，调用了fork（），跟踪代码时我们发现，fork代码执行了两次，第一次，跳过init（）直接执行了for(;;) pause()，第二次执行fork代码后，执行了init（）。奇怪的是，我们在代码中并没有看见向后的goto语句，也没有看到循环语句，是什么原因导致反复执行？请说明理由，并给出代码证据。

首先在copy\_process（）函数中，对进程1 做个性化调整设置，调整tss的数据。

Int copy\_process（int nr, long ebp,…）

P92

然后再执行到如下代码：

＃define switch\_to()……

ljmp….

p196

程序在执行到“ljmp ％0/n/t”这一行，ljmp通过cpu任务们机制自动将进程1 的tss值恢复给cpu,自然也将其中tss.eip恢复给cpu,现在cpu指向fork的if(\_res >=0)这一行。

而此时的\_res值就是进程1中tss的eax的值，这个值在前面被写死为0，即p->tss.eax=0;所以直行到return (type) \_res这一行返回值为0.

Voidmain（void）

｛

if（！fork（））

｛

init（）；

｝

｝

返回后，执行到main函数中if(!fork())这一行，!0值为真，调用init（）函数。

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。

copy\_process执行时因为进程调用了fork函数，会导致中断，中断使CPU硬件自动将SS、ESP、EFLAGS、CS、EIP这几个寄存器的值按照顺序压入 进程0内核栈，又因为函数专递参数是使用栈的，所以刚好可以做为copy\_process的最后五项参数。

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

因为\_syscall0(int,fork)展开是一个真函数，普通真函数调用事需要将eip入栈，返回时需要讲eip出栈。inline是内联函数，它将标明为inline的函数代码放在符号表中，而此处的fork函数需要调用两次，加上inline后先进行词法分析、语法分析正确后就地展开函数，不需要有普通函数的call\ret等指令，也不需要保持栈的eip，效率很高。若不加上inline，第一次调用fork结束时将eip 出栈，第二次调用返回的eip出栈值将是一个错误值。

20.根据代码详细说明copy\_process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。

21.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。

22.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再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（pg3+4092指向的位置）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（pg3-4+4092）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。

Head.s中：（P39）

 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的挂接

movl $pg3+4092,%edi

 movl $0xfff007,%eax  /\*  16Mb - 4096 + 7 (r/w user,p) \*/

 std

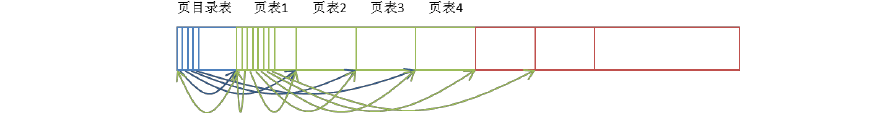
1: stosl   /\* fill pages backwards - more efficient :-) \*/

 subl $0x1000,%eax

jge 1b

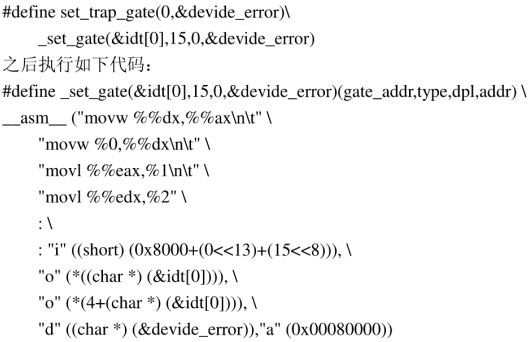
完成页表项与页面的挂接，是从高地址向低地址方向完成挂接的，16M内存全部完成挂接（注意页表从0开始，页表0-页表3）

图见P39



23.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。

对中断描述符表的初始化，就是将异常处理一类的中断服务程序与中断描述符表进行挂接。以set\_trap\_gate(0,&divide\_error)为例，0就表示该中断函数的地址挂接在中断描述符表的第0项位置处，而&devide\_error就是该异常处理函数的地址，对set\_trap\_gate(0,&divide\_error)进行宏展开后得到



%0=0x8f00，%1指向idt[0]的起始地址，%2指向四个字节之后的地址处。

#1、将地址&devide\_error放在EAX的低两个字节，EAX的高两字节不变。#2把0x8f00放入EDX的低两字节，高两字节保持不变。#3、把EAX放在%1所指的地址处，占四字节。#4、将EDX放在%2所指的地址处，占四字节。

24.进程0 fork进程1之前，为什么先要调用move\_to\_user\_mode()？用的是什么方法？解释其中的道理。（P78-79）

因为在Linux-0.11中，除进程0之外，所有进程都是由一个已有进程在用户态下完成创建的。但是此时进程0还处于内核态，因此要调用move\_to\_user\_mode()函数，模仿中断返回的方式，实现进程0的特权级从内核态转化为用户态。又因为在Linux-0.11中，转换特权级时采用中断和中断返回的方式，调用系统中断实现从3到0的特权级转换，中断返回时转换为3特权级。因此，进程0从0特权级到3特权级转换时采用的是模仿中断返回。

25.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。（P89 91 92P97-98 ）

第一个设置进程1的tasks\_truct，另外设置了进程1的堆栈段  
sched.c  
 struct task\_struct \*current = &(init\_task.task);  
fork.c  
 p = (struct task\_struct \*) get\_free\_page();  
 …  
 \*p = \*current;  
 p->tss.esp0 = PAGE\_SIZE + (long) p;  
其中\*current 即为进程0的task结构，在copy\_process中，先复制进程0的task\_struct，然  
后再对其中的值进行修改。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。  
第二个为进程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）。  
copy\_mem  
 …  
 if (copy\_page\_tables(old\_data\_base,new\_data\_base,data\_limit)){  
 …  
其中， copy\_page\_tables 内部  
 …  
 from\_page\_table = (unsigned long \*) (0xfffff000 & \*from\_dir);  
 if (!(to\_page\_table = (unsigned long \*) get\_free\_page()))  
 …  
 for ( ; nr-- > 0 ; from\_page\_table++,to\_page\_table++) {  
 … }  
获取了新的页，且从 from\_page\_table 将页表值拷贝到 to\_page\_table 处。

26.在IA-32中，有大约20多个指令是只能在0特权级下使用，其他的指令，比如cli，并没有这个约定。奇怪的是，在Linux0.11中，在3特权级的进程代码并不能使用cli指令，会报特权级错误，这是为什么？请解释并给出代码证据。

根据Intel Manual，cli和sti指令与CPL和EFLAGS[IOPL]有关。  
CLI：如果 CPL 的权限高于等于eflags中的IOPL的权限，即数值上：cpl <= IOPL，则IF 位清  
除为0；否则它不受影响。EFLAGS 寄存器中的其他标志不受影响。  
#GP(0) –如果CPL大于（特权更小）当前程序或过程的 IOPL，产生保护模式异常。  
由于在内核IOPL的值初始时为0，且未经改变。进程0在move\_to\_user\_mode中，继承了内核的eflags。  
move\_to\_user\_mode()  
…  
"pushfl\n\t" \  
…  
"iret\n" \  
而进程1再copy\_process中，在进程的TSS中，设置了eflags中的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。

27.根据代码详细分析操作系统是如何获得一个空闲页的。（P89-90）

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。 代码如下（ get\_free\_page）  
unsigned long get\_free\_page(void)  
64 {  
65 register unsigned long \_\_res asm("ax");  
66  
67 \_\_asm\_\_("std ; repne ; scasb\n\t" //反向扫描串,al(0)与di不等则重复  
68 "jne 1f\n\t" //找不到空闲页跳转1   
69 "movb $1,1(%%edi)\n\t" //将1付给edi+1的位置，在mem\_map中将找到0的项引用计 ------------------------------------------数置为1  
70 "sall $12,%%ecx\n\t" //ecx算数左移12位，页的相对地址  
71 "addl %2,%%ecx\n\t" //LOW MEN +ecx页物理地址  
72 "movl %%ecx,%%edx\n\t"  
73 "movl $1024,%%ecx\n\t"  
74 "leal 4092(%%edx),%%edi\n\t" //将edx+4kb的有效地址赋给edi  
75 "rep ; stosl\n\t" //将eax赋给edi指向的地址，目的是页面清零。  
76 " movl %%edx,%%eax\n"  
77 "1: cld"  
78 :"=a" (\_\_res)  
79 :"0" (0),"i" (LOW\_MEM),"c" (PAGING\_PAGES),  
80 "D" (mem\_map+PAGING\_PAGES-1) //edx,mem\_map[]的嘴鸥一个元素  
81 );  
82 return \_\_res;  
83 }

28、用户进程自己设计一套LDT表，并与GDT挂接，是否可行，为什么？（P259）

不可行。首先，用户进程不可以设置GDT、LDT，因为Linux0.11将GDT、LDT这两个数据结构设置在内核数据区，是0特权级的，只有0特权级的额代码才能修改设置GDT、LDT；而且，用户也不可以在自己的数据段按照自己的意愿重新做一套GDT、LDT，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，但是真正起作用的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，其他数据结构就算起名字叫GDT、LDT，CPU也一概不认；另外，用户进程也不能将自己制作的GDT、LDT挂接到GDRT、LDRT上，因为对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。

29、保护模式下，线性地址到物理地址的转化过程是什么？（P260-261）

在保护模式下，先行地址到物理地址的转化是通过内存分页管理机制实现的。其基本原理是将整个线性和物理内存区域划分为4K大小的内存页面，系统以页为单位进行分配和回收。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，线性地址的前十位表示也目录表中的页目录项，由此得到所在的页表地址。21~12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。

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

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。

31、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？（P272）

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。

32、详细分析一个进程从创建、加载程序、执行、退出的全过程。

1. 创建进程，调用fork函数。
   1. 准备阶段，为进程在task[64]找到空闲位置，即find\_empty\_process（）；
   2. 为进程管理结构找到储存空间：task\_struct和内核栈。
   3. 父进程为子进程复制task\_struct结构
   4. 复制新进程的页表并设置其对应的页目录项
   5. 分段和分页以及文件继承。
   6. 建立新进程与全局描述符表（GDT）的关联
   7. 将新进程设为就绪态
2. 加载进程
   1. 检查参数和外部环境变量和可执行文件
   2. 释放进程的页表
   3. 重新设置进程的程序代码段和数据段
   4. 调整进程的task\_struct
3. 进程运行
   1. 产生缺页中断并由操作系统响应
   2. 为进程申请一个内存页面
   3. 将程序代码加载到新分配的页面中
   4. 将物理内存地址与线性地址空间对应起来
   5. 不断通过缺页中断加载进程的全部内容
   6. 运行时如果进程内存不足继续产生缺页中断，
4. 进程退出
   1. 进程先处理退出事务
   2. 释放进程所占页面
   3. 解除进程与文件有关的内容并给父进程发信号
   4. 进程退出后执行进程调度

33、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。

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。

34、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？

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，则就是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。

当两个进程共享了一个页面，即R/w为1，导致该页面设为“只读”属性，当其中一个进程需要压栈时就会引发页写保护中断。当页写保护中断产生时，系统会为进程申请新页面，并把原页面内容复制到新页面里。

35、为什么要设计缓冲区，有什么好处？（P310）

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。

缓冲区的好处主要有两点：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对块设备的文件操作运行效率更高。

36、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？

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 是缓冲块的数据内容。

b\_dev和b\_blocknr两个字段把缓冲块和硬盘数据块的关系绑定，同时根据b\_count决定是否废除旧缓冲块而新建缓冲块以保证数据在缓冲区停留时间尽量长。

b\_dev为设备标示，b\_blocknr标示block块好。b\_count用于记录缓冲块被多少个进程共享了。

b\_uptodate和b\_dirt用以保证缓冲块和数据块的正确性。b\_uptodate为1说明缓冲块的数据就是数据块中最新的，进程可以共享缓冲块中的数据。b\_dirt为1时说明缓冲块数据已被进程修改，需要同步到硬盘上。

b\_lock为1时说明缓冲块与数据块在同步数据，此时内核会拦截进程对该缓冲块的操作，直到交互结束才置0。b\_wait用于记录因为b\_lock=1而挂起等待缓冲块的进程数。

+1. 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

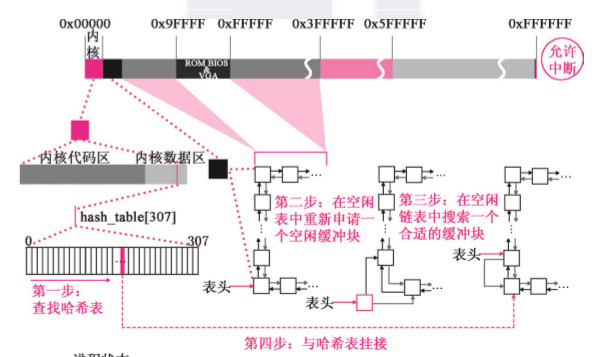
getblk(int dev, int block) 🡪 get\_hash\_table(dev,block) -> find\_buffer(dev,block) -> hash(dev, block)

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



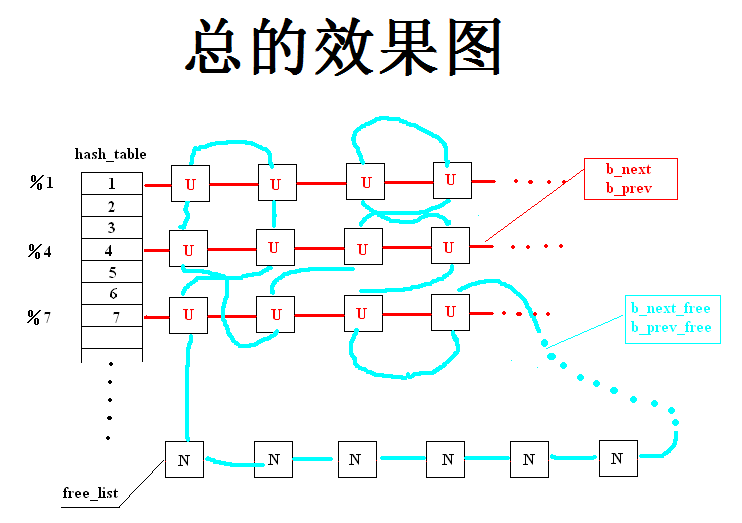
B

//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

}

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，然后从哈西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);

+2. 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\;

+3. 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。

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;

+4. 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。

+5在Linux操作系统中大量的使用了中断、异常类的处理，为什么，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。这种32位中断服务体系是为适应一种被动响应中断信号而建立的。这样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“被动模式”代替“主动轮询”模式来处理终端问题是现在操作系统之所以称之为“现代”的一个重要标志。