操作系统思考题

0 特权
3 特权
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。

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？
答：.....

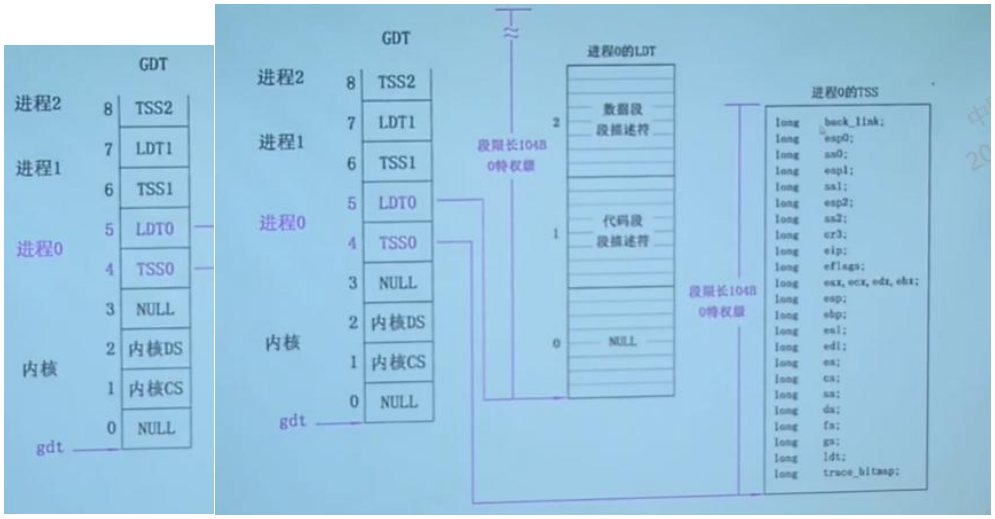
3.为什么 BIOS 把 bootsect 加载到 0x07c00，而不是 0x00000？加载后又马上挪到 0x90000 处，是何道理？为什么不一次加载到位？
答：0x07c00 是“两头约定”“定位识别”中要求的从启动扇区都要被加载到指定位置 0x07c00.只要接到启动操作系统的命令，BIOS 都要“被迫”做这样的工作，而 0x00000 是 BIOS 存放中断向量的地方
加载后又马上挪到 0x90000 是操作系统开始根据自己的需要安排内存。
不加载到位的原因是为了适配不同的操作系统，BIOS 并不知道加载的是哪个操作系统以及后续的内存规划，所以按照约定加载到 0x07c00 即可，后续由操作系统接管。

前置汇编知识：
.text 伪指令 表示接下来的内容为执行程序
.global 伪指令 表示当前汇编文件之外可见的值
a:这种表示一种别名

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

前置知识：Segment 是一种内存管理方式，这种管理方式将内存划分为逻辑段。现代操作系统越来越多使用分页内存管理。使用分段内存管理方式的有：保护模式下的 x86 架构
段选择子是多少位的？
逻辑段 地址 =====映射========> 真实物理地址
jmpi = jump Intersegment 段间跳转 Intel x86 的汇编指令，指令含义为：
jmpi 段内偏移，段选择子
段选择子是根据权限序号如 0x8,0x10 来对应段描述符表中的表项
段描述符表分为两个：全局段描述符表 局部段描述符表
GDT [0] = NULL
GDT [1] = 内核代码段 段基址 段限长 中断代码也在这个段里
GDT 最多 64 个进程，见下表，每个进程是两个段，一个 TSS，一个 LDT
TSS 里面是寄存器，是进程切换的时候的段面，作用是在切换了之后把 cpu 的内容放在内存

切换回来的时候从内存里面读 cpu 的内容



内核 CS：jmpi 0，8 的 8
内核 DS：0x10

TSS:现场保护 存在内存中的那一段
LDT：
每个进程都有一个 description table

将段选择子放入 CS，段内偏移放入 IP
CS:IP
CS: Code Segment Pointer
IP: Instruction Pointer
在实模式下，CS 本身就是代码段基址；但在保护模式下，CS 是代码段选择符
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。

前置知识：
如何判断是否进入保护模式？
CPU 工作方式由实模式转变为保护模式的一个重要的特征就是根据 GDT 决定程序执行。

setup 做了什么？setup 的 jumpi 0,8 是保护模式下的吗？
setup 挪动 system，关中断，设置 IDTR、GDTR，并且将 GDTR 和表中内容关联，打开 A20 扩充寻址位，8259A 可编程中断控制映射，（CPU 里面的）CR0 置 1 调至保护模式（0 是实模式）
setup 的 jumpi 0,8 是保护模式下的。

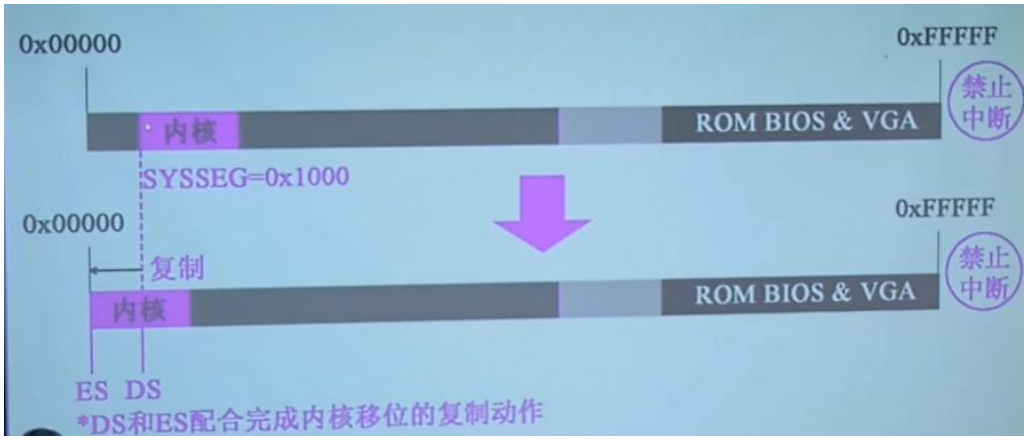
这句代码使 CS 和 GDT 的第 2 项关联，并且使代码段基址指向 0x00000000（32 位）
因为这条指令已经根据 GDT 去决定程序执行，而不是 CS 表示实模式下的代码段基址。
（书本 P27：在实模式下，CS 是代码段基址，在保护模式下，CS 是代码段选择符）[选择子和选择符是同一个东西]

head 做了什么？
head 做的事情都是为了适应保护模式做准备。包括：将寄存器 DS、ES、FS、GS 等从实模式转变为保护模式、重建 GDT、检查 A20 是否打开、建立内核分页机制、ret 实现调用 main 函数

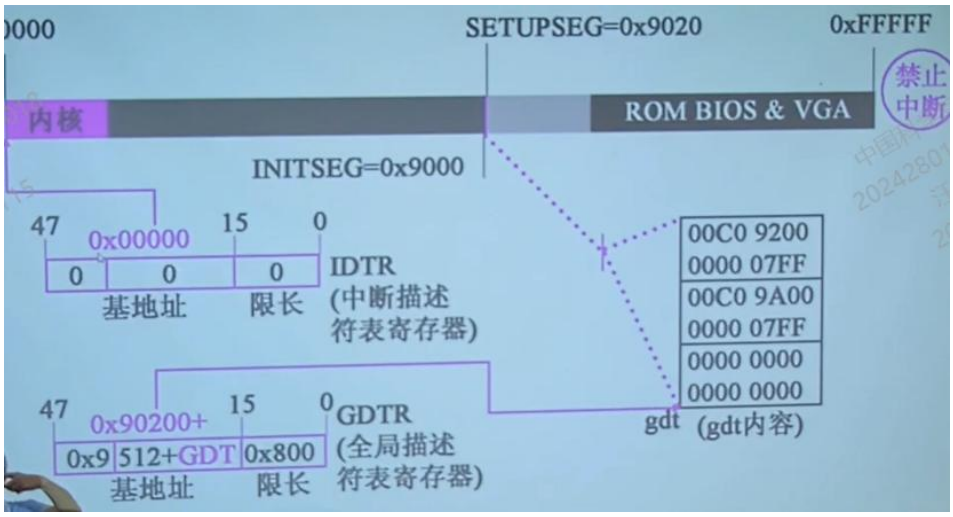
实模式：16 位 没有特权级没有分页 寻址没有虚拟层概念
保护模式：32 位 内存是 4G 线性地址 物理地址 段在线性地址上 页在物理地址上
段描述符描述什么？段基址、段限长、特权级

cli 关闭的是实模式下 16 位的中断，是 BIOS 一加载的时候的顶到头的中断向量表，这个顶头的位置是 BIOS 的硬性规定
cli 关闭的意义是防止在实模式向保护模式转变的过程中有中断到来。main 中 sti 开中断
关中断怎么关？EFLAGS 的 IF 位设置为 0
32 位的中断更加复杂：要完成接续访问，中断描述符表 PL，要融入特权级

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？
保护模式基于段来保护内存。它的保护体现在：



setup 将 system 模块挪到顶头，废除 BIOS 的中断向量表



这里有 IDTR，中断描述符表寄存器，存的是中断描述符表的开始位置，当然也不需要 BIOS 那样一定要放在顶头位置。
GDTR 全局段描述符表寄存器
实模式下的段最多管 64K
保护模式是有段头+限长

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：
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。

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。

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。

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。

1、计算内核代码段、数据段的段基址、段限长、特权级。
内核代码段 CS jumpi 0,8
内核数据段 DS 0x10

2、计算进程 0 的代码段、数据段的段基址、段限长、特权级。
代码段

所有进程的 CS DS 是一样的
内核和进程 0 的代码段是一样的

7、分析 get_free_page()函数的代码，叙述在主内存中获取一个空闲页的技术路线。

8、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。（第 3 章 进程 1 的创建以及执行）

9、详细分析 Linux 操作系统如何设置保护模式的中断机制。

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。

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，却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。

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。

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。

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。

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、#define switch_to(n) {\
struct {long a,b;} __tmp; \
__asm__("cmpl %%ecx,_current\n\t" \
"je 1f\n\t" \
"movw %%dx,%1\n\t" \
"xchgl %%ecx,_current\n\t" \
"ljmp %0\n\t" \
"cmpl %%ecx,_last_task_used_math\n\t" \
"jne 1f\n\t" \
"clts\n" \
"1:" \
::"m" (*&__tmp.a),"m" (*&__tmp.b), \
"d" (_TSS(n)),"c" ((long) task[n])); \
}
代码中的"ljmp %0\n\t" 很奇怪，按理说 jmp 指令跳转到得位置应该是一条指令的地址，可是这行代码却跳到了"m" (*&__tmp.a)，这明明是一个数据的地址，更奇怪的，这行代码竟然能正确执行。请论述其中的道理。

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。

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

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？

第二章有个 panic：一旦调用 panic 函数说明出现大问题，函数作用：死机
比死机更可怕的事情：如核电站爆炸等等

10、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？

11、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。

12、wait_on_buffer 函数中为什么不用 if（）而是用 while（）？

13、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。

问题：
一共有几题？
内核的线性寻址空间是 16MB 吗

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？

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？给出代码证据。

3、wait_on_buffer 函数中为什么不用 if（）而是用 while（）？

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。

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。

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？