Linux操作系统的内核引导程序详细解析 PDF转换可能丢失图片或格式，建议阅读原文 https://www.100test.com/kao_ti2020/268/2021_2022_Linux_E6_93_8D_E4_BD_c67_268981.htm 这段程序是Linux操作系统启动boot程序! bootsect.s (c) 1991, 1992 Linus Torvalds 版权所有 ! Drew Eckhardt修改过 ! Bruce Evans (bde)修改过 ! ! bootsect.s 被bios-启动子程序加载至0x7c00 (31k)处，并将自己 ! 移到了地址0x90000 (576k)处，并跳转至那里。 ! ! bde - 不能盲目地跳转，有些系统可能只有512k的低 ! 内存。使用中断0x12来获得(系统的)最高内存、等。 ! ! 它然后使用BIOS中断将setup直接加载到自己的后面(0x90200)(576.5k)， ! 并将系统加载到地址0x10000处。 ! ! 注意! 目前的内核系统最大长度限制为(8*65536-4096)(508k)字节长，即使是在 ! 将来这也是没有问题的。我想让它保持简单明了。这样508k的最大内核长度应该 ! 是足够了，尤其是这里没有象minix中一样包含缓冲区高速缓冲(而且尤其是现在 ! 内核是压缩的 :-) ! ! 加载程序已经做的尽量地简单了，所以持续的读出错将导致死循环。只能手工重启。 ! 只要可能，通过一次取得整个磁道，加载过程可以做的很快的。 #include /* 为取得CONFIG_ROOT_RDONLY参数 */ !! config.h中(即autoconf.h中)没有CONFIG_ROOT_RDONLY定义!!!? #include .text SETUPSECS = 4 ! 默认的setup程序扇区数(setup-sectors)的默认值. BOOTSEG = 0x7C0 ! bootsect的原始地址. INITSEG = DEF_INITSEG ! 将bootsect程序移到这个段处(0x9000) - 避开. SETUPSEG = DEF_SETUPSEG ! 设置程序(setup)从这里开始(0x9020). SYSSEG = DEF_SYSSEG ! 系统加载

至0x1000(65536)(64k)段处. SYSSIZE = DEF_SYSSIZE ! 系统的大小(0x7F00): 要加载的16字节为一节的数. !! 以上4个DEF_参数定义在boot.h中: !! DEF_INITSEG 0x9000 !! DEF_SYSSEG 0x1000 !! DEF_SETUPSEG 0x9020 !! DEF_SYSSIZE 0x7F00 (=32512=31.75k)*16=508k ! ROOT_DEV &amp. SWAP_DEV 现在是由"build"中编制的. ROOT_DEV = 0 SWAP_DEV = 0 #ifndef SVGA_MODE #define SVGA_MODE ASK_VGA #endif #ifndef RAMDISK #define RAMDISK 0 #endif #ifndef CONFIG_ROOT_RDONLY #define CONFIG_ROOT_RDONLY 1 #endif ! ld86 需要一个入口标识符，这和通常的一样. .globl _main _main: #if 0 /* 调试程序的异常分支，除非BIOS古怪（比如老的HP机）否则是无害的 */ int 3 #endif mov ax,#BOOTSEG !! 将ds段寄存器置为0x7C0. mov ds,ax mov ax,#INITSEG !! 将es段寄存器置为0x9000. mov es,ax mov cx,#256 !! 将cx计数器置为256(要移动256个字, 512字节). sub si,si !! 源地址ds:si=0x07C0:0x0000. sub di,di !! 目的地址es:di=0x9000:0x0000. cld !! 清方向标志. rep !! 将这段程序从0x7C0:0(31k)移至0x9000:0(576k)处. movsw !! 共256个字(512字节)(0x200长). jmpi go,INITSEG !! 间接跳转至移动后的本程序go处. ! ax和es现在已经含有INITSEG的值(0x9000). go: mov di,#0x4000-12 ! 0x4000(16k)是>=bootsect setup 的长度 ! 堆栈的长度 的任意的值. ! 12 是磁盘参数块的大小 es:di=0x94000-12=592k-12. ! bde - 将0xff00改成了0x4000以从0x6400处使用调试程序(bde)。如果 ! 我们检测过最高内存的话就不用担心这事了，还有，我的BIOS可以被配置为将wini驱动表 ! 放在内存高端而不是放在向量表中。老式的堆栈区可能会搞乱驱动表. mov ds,ax ! 置ds

数据段为0x9000. mov ss,ax ! 置堆栈段为0x9000. mov sp,di ! 置堆栈指针INITSEG:0x4000-12处. /* * 许多BIOS的默认磁盘参数表将不能 * 进行扇区数大于在表中指定 * 的最大扇区数(- 在某些情况下 * 这意味着是7个扇区)后面的多扇区的读操作。 * * 由于单个扇区的读操作是很慢的而且当然是没问题的， * 我们必须在RAM中(为第一个磁盘)创建新的参数表。 * 我们将把最大扇区数设置为36 - 我们在一个ED 2.88驱动器上所能 * 遇到的最大值。 * * 此值太高是没有任何害处的，但是低的话就会有问题了。 * * 段寄存器是这样的: ds=es=ss=cs - INITSEG,(=0X9000) * fs = 0, gs没有用到。 */ ! 上面执行重复操作(rep)以后， cx为0. mov fs,cx !! 置fs段寄存器=0. mov bx,#0x78 ! fs:bx是磁盘参数表的地址. push ds seg fs lds si,(bx) ! ds:si是源地址. !! 将fs:bx地址所指的指针值放入ds:si中. mov cl,#6 ! 拷贝12个字节到0x9000:0x4000-12开始处. cld push di !! 指针0x9000:0x4000-12处. rep movsw pop di !! di仍指向0x9000:0x4000-12处(参数表开始处). pop si !! ds => si=INITSEG(=0X9000). movb 4(di),*36 ! 修正扇区计数值. seg fs mov (bx),di !! 修改fs:bx(0000:0x0078)处磁盘参数表的地址为0x9000:0x4000-12. seg fs mov 2(bx),es ! 将setup程序所在的扇区(setup-sectors)直接加载到boot块的后面。 !! 0x90200开始处. ! 注意， es已经设置好了。 ! 同样经过rep循环后cx为0 load_setup: xor ah,ah ! 复位软驱(FDC). xor dl,dl int 0x13 xor dx,dx ! 驱动器0, 磁头0. mov cl,#0x02 ! 从扇区2开始， 磁道0. mov bx,#0x0200 ! 置数据缓冲区地址=es:bx=0x9000:0x200. ! 在INITSEG段中,即0x90200处. mov ah,#0x02 ! 要调用功能号2(读操作). mov al,setup_sects ! 要读入的扇区数SETUPSECS=4. ! (

假释所有数据都在磁头0、磁道0). int 0x13 ! 读操作. jnc ok_load_setup ! ok则继续. push ax ! 否则显示出错信息。保存ah的值(功能号2). call print_nl !! 打印换行. mov bp,sp !! bp将作为调用print_hex的参数. call print_hex !! 打印bp所指的数据. pop ax jmp load_setup !! 重试! !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!! !!INT 13 - DISK - READ SECTOR(S) INTO MEMORY !! AH = 02h !! AL = number of sectors to read (must be nonzero) !! CH = low eight bits of cylinder number !! CL = sector number 1-63 (bits 0-5) !! high two bits of cylinder (bits 6-7, hard disk only) !! DH = head number !! DL = drive number (bit 7 set for hard disk) !! ES:BX -> data buffer !! Return: CF set on error !! if AH = 11h (corrected ECC error), AL = burst length !! CF clear if successful !! AH = status (see #00234) !! AL = number of sectors transferred (only valid if CF set for some !! BIOSes) !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!! ok_load_setup: ! 取得磁盘驱动器参数，特别是每磁道扇区数(nr of sectors/track). #if 0 ! bde - Phoenix BIOS手册中提到功能0x08只对硬盘起作用。 ! 但它对于我的一个BIOS(1987 Award)不起作用。 ! 不检查错误码是致命的错误。 xor dl,dl mov ah,#0x08 ! AH=8用于取得驱动器参数. int 0x13 xor ch,ch !!!!!!!!!!!!!!!!!!!!!!!!!! !! INT 13 - DISK - GET DRIVE PARAMETERS (PC,XT286,CONV,PS,ESDI,SCSI) !! AH = 08h !! DL = drive (bit 7 set for hard disk) !!Return: CF set on error !! AH = status (07h) (see #00234) !! CF clear if successful !! AH = 00h !! AL = 00h on at least some BIOSes !! BL = drive type (AT/PS2 floppies only) (see #00242) !! CH = low eight bits of maximum cylinder number !! CL = maximum sector number (bits 5-0) !! high two bits of maximum cylinder number (bits 7-6) !! DH = maximum

head number !! DL = number of drives !! ES:DI -> drive parameter table (floppies only) !!!!!!!!!!!!!!!!!!!!!!!!!!!! #else ! 好象没有BIOS调用可取得扇区数。如果扇区36可以读就推测是36个扇区，！如果扇区18可读就推测是18个扇区，如果扇区15可读就推测是15个扇区，！否则推测是9. [36, 18, 15, 9] mov si,#disksizes ! ds:si-> 要测试扇区数大小的表. probe_loop: lodsb !! ds:si所指的字节 =>al, si=si 1. cbw ! 扩展为字(word). mov sectors, ax ! 第一个值是36，最后一个是9. cmp si,#disksizes 4 jae got_sectors ! 如果所有测试都失败了，就试9. xchg ax,cx ! cx = 磁道和扇区(第一次是36=0x0024). xor dx,dx ! 驱动器0，磁头0. xor bl,bl !! 设置缓冲区es:bx = 0x9000:0x0a00(578.5k). mov bh,setup_sects !! setup_sects = 4 (共2k). inc bh shl bh,#1 ! setup后面的地址(es=cs). mov ax,#0x0201 ! 功能2(读)，1个扇区. int 0x13 jc probe_loop ! 如果不对，就试用下一个值. #endif got_sectors: ! 恢复es mov ax,#INITSEG mov es,ax ! es = 0x9000. ! 打印一些无用的信息(换行后，显示Loading) mov ah,#0x03 ! 读光标位置. xor bh,bh int 0x10 mov cx,#9 mov bx,#0x0007 ! 页0，属性7 (normal). mov bp,#msg1 mov ax,#0x1301 ! 写字符串，移动光标. int 0x10 ! ok, 我们已经显示出了信息，现在 ! 我们要加载系统了(到0x10000处)(64k处) mov ax,#SYSSEG mov es,ax ! es=0x01000的段. call read_it !! 读system，es为输入参数； call kill_motor !! 关闭驱动器马达； call print_nl !! 打印回车换行； ! 这以后，我们来检查要使用哪个根设备(root-device)。如果已指定了设备(!=0) ! 则不做任何事而使用给定的设备。否则的话，使用/dev/fd0H2880 (2,32)或/dev/PS0 (2,28) ! 或者是/dev/at0 (2,8)之一，这取决于我们假设我们知道的扇区数而定。 !! |__ ps0??

(x,y)--表示主、次设备号？ seg cs mov ax,root_dev or ax,ax jne root_defined seg cs mov bx,sectors !! sectors = 每磁道扇区数； mov ax,#0x0208 ! /dev/ps0 - 1.2Mb. cmp bx,#15 je root_defined mov al,#0x1c ! /dev/PS0 - 1.44Mb !! 0x1C = 28. cmp bx,#18 je root_defined mov al,0x20 ! /dev/fd0H2880 - 2.88Mb. cmp bx,#36 je root_defined mov al,#0 ! /dev/fd0 - autodetect. root_defined: seg cs mov root_dev,ax !! 其中保存由设备的主、次设备号；！这以后(所有程序都加载了)，我们就跳转至！被直接加载到boot块后面的setup程序去： jmpi 0,SETUPSEG !! 跳转到0x9020:0000(setup程序的开始位置). ! 这段程序将系统(system)加载到0x10000(64k)处, ! 注意不要跨越64kb边界。我们试图以最快的速度！来加载，只要可能就整个磁道一起读入。 ！！输入(in): es - 开始地址段(通常是0x1000) ! sread: .word 0 ! 当前磁道已读的扇区数； head: .word 0 ! 当前磁头； track: .word 0 ! 当前磁道； read_it: mov al,setup_sects inc al mov sread,al !! 当前sread=5； mov ax,es !! es=0x1000； test ax,#0x0fff !! (ax AND 0x0fff, if ax=0x1000 then zero-flag=1)； die: jne die ! es 必须在64kB的边界； xor bx,bx ! bx 是段内的开始地址； rp_read: #ifdef __BIG_KERNEL__ #define CALL_HIGHLOAD_KLUDGE .word 0x1eff, 0x220 ! 调用 far * bootsect_kludge ! 注意: as86不能汇编这； CALL_HIGHLOAD_KLUDGE ! 这是在setup.S中的程序 ； #else mov ax,es sub ax,#SYSSEG ! 当前es段值减system加载时的启始段值(0x1000)； #endif cmp ax,syssize ! 我们是否已经都加载了？(ax=0x7f00 ?)； jbe ok1_read !! if ax ret !! 全都加载完了，返回！ ok1_read: mov ax,sectors !! sectors=每磁道扇区数； sub ax,sread !! 减去当前磁道已读扇区数，al=当前磁道未读的

扇区数(ah=0)； mov cx,ax shl cx,#9 !! 乘512，cx = 当前磁道未读的字节数； add cx,bx !! 加上段内偏移值，es:bx为当前读入的数据缓冲区地址； jnc ok2_read !! 如果没有超过64K则继续读； je ok2_read !! 如果正好64K也继续读； xor ax,ax sub ax,bx shr ax,#9 ok2_read: call read_track !! es:bx ->缓冲区，al=要读的扇区数，也即当前磁道未读的扇区数； mov cx,ax !! ax仍为调用read_track之前的值，即为读入的扇区数； add ax,sread !! ax = 当前磁道已读的扇区数； cmp ax,sectors !! 已经读完当前磁道上的扇区了吗？ jne ok3_read !! 没有，则跳转； mov ax,#1 sub ax,head !! 当前是磁头1吗？ jne ok4_read !! 不是(是磁头0)则跳转(此时ax=1)； inc track !! 当前是磁头1，则读下一磁道(当前磁道加1)； ok4_read: mov head,ax !! 保存当前磁头号； xor ax,ax !! 本磁道已读扇区数清零； ok3_read: mov sread,ax !! 存本磁道已读扇区数； shl cx,#9 !! 刚才一次读操作读入的扇区数*512； add bx,cx !! 调整数据缓冲区的起始指针； jnc rp_read !! 如果该指针没有超过64K的段内最大偏移量，则跳转继续读操作； mov ax,es !! 如果超过了，则将段地址加0x1000(下一个64K段). add ah,#0x10 mov es,ax xor bx,bx !! 缓冲区地址段内偏移量置零； jmp rp_read !! 继续读操作； read_track: pusha !! 将寄存器ax,cx,dx,bx,sp,bp,si,di压入堆栈； pusha mov ax,#0xe2e ! loading... message 2e = . !! 显示一个. mov bx,#7 int 0x10 popa mov dx,track !! track = 当前磁道； mov cx,sread inc cx !! cl = 扇区号，要读的起始扇区； mov ch,dl !! ch = 磁道号的低8位； mov dx,head !! mov dh,dl !! dh = 当前磁头号； and dx,#0x0100 !! dl = 驱动器号(0)； mov ah,#2 !! 功能2(读)，es:bx指向读数据缓冲区； push dx ! 为出错转储保存寄存器的值到堆栈上； push cx

push bx push ax int 0x13 jc bad_rt !! 如果出错，则跳转； add sp, #8 !! 清(放弃)堆栈上刚推入的4个寄存器值； popa ret bad_rt: push ax ! 保存出错码； call print_all ! ah = error, al = read； xor ah,ah xor dl,dl int 0x13 add sp,#10 popa jmp read_track /* * print_all是用于调试的。 * 它将打印出所有寄存器的值。所作的假设是 * 从一个子程序中调用的，并有如下所示的堆栈帧结构 * dx * cx * bx * ax * error * ret * */ print_all: mov cx,#5 ! 出错码 4个寄存器 mov bp,sp print_loop: push cx ! 保存剩余的计数值 call print_nl ! 为了增强阅读性，打印换行 cmp cl, #5 jae no_reg ! 看看是否需要寄存器的名称 mov ax,#0xe05 A - l sub al,cl int 0x10 mov al,#X int 0x10 mov al,#: int 0x10 no_reg: add bp,#2 ! 下一个寄存器 call print_hex ! 打印值 pop cx loop print_loop ret print_nl: !! 打印回车换行。 mov ax,#0xe0d ! CR int 0x10 mov al,#0xa ! LF int 0x10 ret /* * print_hex是用于调试目的的，打印出 * ss:bp所指向的十六进制数。 * !! 例如，十六进制数是0x4321时，则al分别等于4,3,2,1调用中断打印出来 4321 */ print_hex: mov cx, #4 ! 4个十六进制数字 mov dx, (bp) ! 将(bp)所指的值放入dx中 print_digit: rol dx, #4 ! 循环以使低4比特用上 !! 取dx的高4比特移到低4比特处。 mov ax, #0xe0f ! ah = 请求的功能值，al = 半字节(4个比特)掩码。 and al, dl !! 取dl的低4比特值。 add al, #0x90 ! 将al转换为ASCII十六进制码(4个指令) daa !! 十进制调整 adc al, #0x40 !! (adc dest, src ==> dest := dest src c) daa int 0x10 loop print_digit ret /* * 这个过程(子程序) 关闭软驱的马达，这样 * 我们进入内核后它的状态就是已知的，以后也就 * 不用担心它了。 */ kill_motor: push dx mov dx,#0x3f2 xor al,al outb pop dx ret !! 数据区 sectors: .word 0 !! 当前每磁道扇区数

。(36||18||15||9) disksizes: !! 每磁道扇区数表 .byte 36, 18, 15, 9 msg1: .byte 13, 10 .ascii "Loading" .org 497 !! 从boot程序的二进制文件的497字节开始 setup_sects: .byte SETUPSECS root_flags: .word CONFIG_ROOT_RDONLY syssize: .word SYSSIZE swap_dev: .word SWAP_DEV ram_size: .word RAMDISK vid_mode: .word SVGA_MODE root_dev: .word ROOT_DEV boot_flag: !! 分区启动标志 .word 0xAA55