```
!
! SYS_SIZE is the number of clicks (16 bytes) to be loaded.
! 0x7F00 is 0x7F000 bytes = 508kB, more than enough for current
! versions of linux which compress the kernel
!
#include <linux/config.h>
SYSSIZE = DEF_SYSSIZE
!
!	bootsect.s		Copyright (C) 1991, 1992 Linus Torvalds
!	modified by Drew Eckhardt
!	modified by Bruce Evans (bde)
!
! bootsect.s is loaded at 0x7c00 by the bios-startup routines, and moves
! itself out of the way to address 0x90000, and jumps there.
!
! bde - should not jump blindly, there may be systems with only 512K low
! memory.  Use int 0x12 to get the top of memory, etc.
!
! It then loads 'setup' directly after itself (0x90200), and the system
! at 0x10000, using BIOS interrupts.
!
! NOTE! currently system is at most (8*65536-4096) bytes long. This should
! be no problem, even in the future. I want to keep it simple. This 508 kB
! kernel size should be enough, especially as this doesn't contain the
! buffer cache as in minix (and especially now that the kernel is
! compressed :-)
!
! The loader has been made as simple as possible, and continuos
! read errors will result in a unbreakable loop. Reboot by hand. It
! loads pretty fast by getting whole tracks at a time whenever possible.

.text

SETUPSECS = 4                       ! nr of setup-sectors
BOOTSEG   = 0x07C0                  ! original address of boot-sector
INITSEG   = DEF_INITSEG                     ! we move boot here - out of the way
SETUPSEG  = DEF_SETUPSEG            ! setup starts here
SYSSEG    = DEF_SYSSEG              ! system loaded at 0x10000 (65536).

! ROOT_DEV & SWAP_DEV are now written by "build".
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 0
#endif

! ld86 requires an entry symbol. This may as well be the usual one.
.globl _main
_main:
#if 0 /* hook for debugger, harmless unless BIOS is fussy (old HP) */
	int	3
#endif
	mov	ax,#BOOTSEG
	mov	ds,ax
	mov	ax,#INITSEG
	mov	es,ax
	mov	cx,#256
	sub	si,si
	sub	di,di
	cld
	rep
	movsw
	jmpi	go,INITSEG

go:	mov	ax,cs
	mov	dx,#0x4000-12	! 0x4000 is arbitrary value >= length of
				! bootsect + length of setup + room for stack
				! 12 is disk parm size
```

```
! bde - changed 0xff00 to 0x4000 to use debugger at 0x6400 up (bde).  We
! wouldn't have to worry about this if we checked the top of memory.  Also
! my BIOS can be configured to put the wini drive tables in high memory
! instead of in the vector table.  The old stack might have clobbered the
! drive table.

        mov     ds,ax
        mov     es,ax
        mov     ss,ax           ! put stack at INITSEG:0x4000-12.
        mov     sp,dx
/*
 *      Many BIOS's default disk parameter tables will not
 *      recognize multi-sector reads beyond the maximum sector number
 *      specified in the default diskette parameter tables - this may
 *      mean 7 sectors in some cases.
 *
 *      Since single sector reads are slow and out of the question,
 *      we must take care of this by creating new parameter tables
 *      (for the first disk) in RAM.  We will set the maximum sector
 *      count to 18 - the most we will encounter on an HD 1.44.
 *
 *      High doesn't hurt.  Low does.
 *
 *      Segments are as follows: ds=es=ss=cs - INITSEG,
 *              fs = 0, gs = parameter table segment
 */

        push    #0
        pop     fs
        mov     bx,#0x78                ! fs:bx is parameter table address
        seg fs
        lgs     si,(bx)                 ! gs:si is source

        mov     di,dx                   ! es:di is destination
        mov     cx,#6                   ! copy 12 bytes
        cld

        rep
        seg gs
        movsw

        mov     di,dx
        movb    4(di),*18               ! patch sector count

        seg fs
        mov     (bx),di
        seg fs
        mov     2(bx),es

        mov     ax,cs
        mov     fs,ax
        mov     gs,ax

        xor     ah,ah                   ! reset FDC
        xor     dl,dl
        int     0x13

! load the setup-sectors directly after the bootblock.
! Note that 'es' is already set up.

load_setup:
        xor     dx, dx                  ! drive 0, head 0
        mov     cx,#0x0002              ! sector 2, track 0
        mov     bx,#0x0200              ! address = 512, in INITSEG
        mov     ax,#0x0200+SETUPSECS    ! service 2, nr of sectors
                                        ! (assume all on head 0, track 0)
        int     0x13                    ! read it
        jnc     ok_load_setup           ! ok - continue

        push    ax                      ! dump error code
        call    print_nl
        mov     bp, sp
        call    print_hex
        pop     ax

        xor     dl, dl                  ! reset FDC
        xor     ah, ah
        int     0x13
        jmp     load_setup
```

```
ok_load_setup:

! Get disk drive parameters, specifically nr of sectors/track

#if 0

! bde - the Phoenix BIOS manual says function 0x08 only works for fixed
! disks.  It doesn't work for one of my BIOS's (1987 Award).  It was
! fatal not to check the error code.

        xor     dl,dl
        mov     ah,#0x08              ! AH=8 is get drive parameters
        int     0x13
        xor     ch,ch
#else

! It seems that there is no BIOS call to get the number of sectors.  Guess
! 18 sectors if sector 18 can be read, 15 if sector 15 can be read.
! Otherwise guess 9.

        xor     dx, dx                ! drive 0, head 0
        mov     cx,#0x0012            ! sector 18, track 0
        mov     bx,#0x0200+SETUPSECS*0x200  ! address after setup (es = cs)
        mov     ax,#0x0201            ! service 2, 1 sector
        int     0x13
        jnc     got_sectors
        mov     cl,#0x0f              ! sector 15
        mov     ax,#0x0201            ! service 2, 1 sector
        int     0x13
        jnc     got_sectors
        mov     cl,#0x09

#endif

got_sectors:
        seg cs
        mov     sectors,cx
        mov     ax,#INITSEG
        mov     es,ax

! Print some inane message

        mov     ah,#0x03              ! read cursor pos
        xor     bh,bh
        int     0x10

        mov     cx,#9
        mov     bx,#0x0007            ! page 0, attribute 7 (normal)
        mov     bp,#msg1
        mov     ax,#0x1301            ! write string, move cursor
        int     0x10

! ok, we've written the message, now
! we want to load the system (at 0x10000)

        mov     ax,#SYSSEG
        mov     es,ax          ! segment of 0x010000
        call    read_it
        call    kill_motor
        call    print_nl

! After that we check which root-device to use. If the device is
! defined (!= 0), nothing is done and the given device is used.
! Otherwise, either /dev/PS0 (2,28) or /dev/at0 (2,8), depending
! on the number of sectors that the BIOS reports currently.

        seg cs
        mov     ax,root_dev
        or      ax,ax
        jne     root_defined
        seg cs
        mov     bx,sectors
        mov     ax,#0x0208            ! /dev/ps0 - 1.2Mb
        cmp     bx,#15
        je      root_defined
        mov     ax,#0x021c            ! /dev/PS0 - 1.44Mb
        cmp     bx,#18
        je      root_defined
        mov     ax,#0x0200            ! /dev/fd0 - autodetect
```

```
root_defined:
        seg cs
        mov     root_dev,ax

! after that (everyting loaded), we jump to
! the setup-routine loaded directly after
! the bootblock:

        jmpi    0,SETUPSEG

! This routine loads the system at address 0x10000, making sure
! no 64kB boundaries are crossed. We try to load it as fast as
! possible, loading whole tracks whenever we can.
!
! in:   es - starting address segment (normally 0x1000)
!
sread:  .word 1+SETUPSECS      ! sectors read of current track
head:   .word 0                ! current head
track:  .word 0                ! current track

read_it:
        mov ax,es
        test ax,#0x0fff
die:    jne die                ! es must be at 64kB boundary
        xor bx,bx              ! bx is starting address within segment
rp_read:
        mov ax,es
        sub ax,#SYSSEG
        cmp ax,syssize         ! have we loaded all yet?
        jbe ok1_read
        ret
ok1_read:
        seg cs
        mov ax,sectors
        sub ax,sread
        mov cx,ax
        shl cx,#9
        add cx,bx
        jnc ok2_read
        je ok2_read
        xor ax,ax
        sub ax,bx
        shr ax,#9
ok2_read:
        call read_track
        mov cx,ax
        add ax,sread
        seg cs
        cmp ax,sectors
        jne ok3_read
        mov ax,#1
        sub ax,head
        jne ok4_read
        inc track
ok4_read:
        mov head,ax
        xor ax,ax
ok3_read:
        mov sread,ax
        shl cx,#9
        add bx,cx
        jnc rp_read
        mov ax,es
        add ah,#0x10
        mov es,ax
        xor bx,bx
        jmp rp_read

read_track:
        pusha
        pusha
        mov     ax, #0xe2e     ! loading... message 2e = .
        mov     bx, #7
        int     0x10
        popa

        mov     dx,track
        mov     cx,sread
        inc     cx
        mov     ch,dl
```

```
        mov     dx,head
        mov     dh,dl
        and     dx,#0x0100
        mov     ah,#2

        push    dx                              ! save for error dump
        push    cx
        push    bx
        push    ax

        int     0x13
        jc      bad_rt
        add     sp, #8
        popa
        ret

bad_rt: push    ax                              ! save error code
        call    print_all                       ! ah = error, al = read


        xor ah,ah
        xor dl,dl
        int 0x13


        add     sp, #10
        popa
        jmp read_track

/*
 *      print_all is for debugging purposes.
 *      It will print out all of the registers.  The assumption is that this is
 *      called from a routine, with a stack frame like
 *      dx
 *      cx
 *      bx
 *      ax
 *      error
 *      ret <- sp
 *
*/

print_all:
        mov     cx, #5          ! error code + 4 registers
        mov     bp, sp

print_loop:
        push    cx              ! save count left
        call    print_nl        ! nl for readability

        cmp     cl, 5
        jae     no_reg          ! see if register name is needed

        mov     ax, #0xe05 + 'A - 1
        sub     al, cl
        int     0x10

        mov     al, #'X
        int     0x10

        mov     al, #':
        int     0x10

no_reg:
        add     bp, #2          ! next register
        call    print_hex       ! print it
        pop     cx
        loop    print_loop
        ret


print_nl:
        mov     ax, #0xe0d      ! CR
        int     0x10
        mov     al, #0xa        ! LF
        int     0x10
        ret
```

```
/*
 *      print_hex is for debugging purposes, and prints the word
 *      pointed to by ss:bp in hexadecmial.
*/

print_hex:
        mov     cx, #4          ! 4 hex digits
        mov     dx, (bp)        ! load word into dx
print_digit:
        rol     dx, #4          ! rotate so that lowest 4 bits are used
        mov     ah, #0xe
        mov     al, dl          ! mask off so we have only next nibble
        and     al, #0xf
        add     al, #'0         ! convert to 0-based digit
        cmp     al, #'9         ! check for overflow
        jbe     good_digit
        add     al, #'A - '0 - 10

good_digit:
        int     0x10
        loop    print_digit
        ret


/*
 * This procedure turns off the floppy drive motor, so
 * that we enter the kernel in a known state, and
 * don't have to worry about it later.
 */
kill_motor:
        push dx
        mov dx,#0x3f2
        xor al, al
        outb
        pop dx
        ret

sectors:
        .word 0

msg1:
        .byte 13,10
        .ascii "Loading"

.org 498
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:
        .word 0xAA55
```