

3.3.7. Boot Linux zImage from File System

The MTX booter can be adapted to booting small Linux kernels from an EXT2 file system. When booting a Linux zImage there is a slight problem. The contents of an image file are stored in (1KB) disk blocks. During booting, we prefer to load the image by blocks. As pointed out earlier, starting from segment 0x1000, loading 1KB blocks will not cross any cylinder or 64KB boundary. In a Linux zImage, the kernel image follows BOOT+SETUP immediately. If the number of BOOT+SETUP sectors is odd, the kernel image does not begin at a block boundary, which makes loading by blocks difficult. For instance, if we load the block that contains the last sector of SETUP and the first sector of kernel to 0x1000-0x20, it would cross 64KB boundary. If we load the first kernel sector to 0x1000, followed by loading blocks, we must monitor the blocks and split up any block that crosses a 64KB boundary. It would be very hard to write such a booter in 1KB. There are two possible ways to deal with this problem. The simplest way is to assume that the number of SETUP sectors is odd, so that BOOT+SETUP=even. If the number of SETUP sectors is even, e.g. 10, we can always pad a dummy sector between SETUP and the kernel image to make the latter begin at a block boundary. Another way is to load the block that contains the last SETUP sector and the first kernel sector to 0x1000, followed by loading blocks. When loading completes, if the number of SETUP sectors is even, we simply move the loaded image downward (address-wise) by one sector. The Linux zImage booter uses the second technique. The booter's C code is shown below. The move(segment) function is in assembly, which is trivial and therefore not shown. The booter size is 1024 bytes, which is still within the 1KB limit.

```
/****** C code of Linux bzImage booter *****/
u16 iblock, NSEC = 2;
char b1[1024], b2[1024], b3[1024]; // b2[ ] and b3[ ] are adjacent
main()
{
    char    *cp, *name[2];
    u16     i, ino, setup, blk, nblk;
    u32     *up;
    INODE   *ip;
    GD      *gp;
    name[0] = "boot"; name[1] = "linux"; // hard coded /boot/linux so far
    getblk(2, b1);
    gp=(GD *)b1; // get group0 descriptor to find inode table start block
    // read inode start block to get root inode
    iblock = (u16)gp->bg_inode_table;
    getblk(iblock, b1);
    ip = (INODE *)b1 + 1; // ip points at root inode
    // search for image file name
    for (i=0; i<2; i++){
        ino = search(ip, name[i]) - 1;
        if (ino < 0) error(); // if search() failed
        getblk(iblock + (ino / 8), b1);
        ip = (INODE *)b1 + (ino % 8);
    }
    // get setup_sectors from linux BOOTsector[497]
    getblk((u16)ip->i_block[0], b2);
    setup = b2[497];
    nblk = (1 + setup)/2; // number of [bootsector+SETUP] blocks
```

```

// read in indirect & double indirect blocks before changing ES
getblk((u16)ip->i_block[12], b2); // get indirect block into b2[ ]
getblk((u16)ip->i_block[13], b3); // get db indirect block into b3[ ]
up = (u32 *)b3;
getblk((u16)*up, b3)           // get first double indirect into b3[ ]
setes(0x9000);                 // loading segment of BOOT+SETUP
for (i=0; i<12; i++){         // nblk of these are bootblock+SETUP
    if (i==nblk){
        if ((setup & 1)==0)    // if setp=even => need 1/2 block more
            getblk((u16)ip->i_block[i], 0);
            setes(0x1000);      // set ES for kernel image at 0x1000
        }
        getblk((u16)ip->i_block[i], 0); // setup=even:1/2 SETUP
        incs();
    }
    //load indirect and double indirect blocks in b2[]b3[]
    up = (u32 *)b2;             // access b2[ ]b3[ ] as u32's
    while(*up++){
        getblk((u16)*up, 0);     // load block to (ES,0)
        incs(); putc('.');
    }
    // finally, if setup is even, move kernel image DOWN one sector
    if ((setup & 1)==0)
        for (i=1; i<9; i++)
            move(i*0x1000); // move one 64 KB segment at a time
}

```

Figure 3.16 shows the screen of booting Linux from a file system, in which each dot represents loading a 1KB disk block of the Linux kernel image.

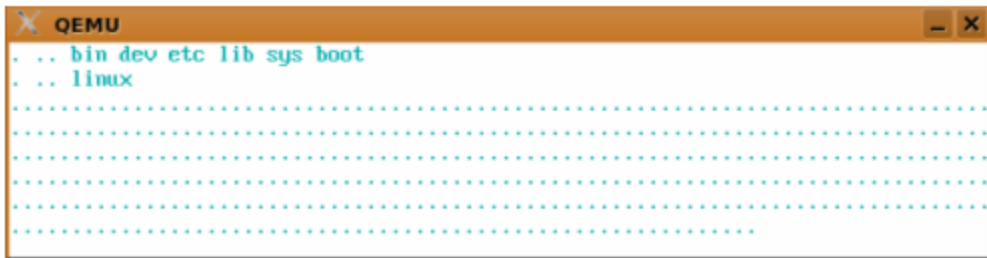


Figure 3.16. Booting Linux from File System

When the Linux kernel boots up, it must mount a root file system in order to run. In the busyboxlinux directory, the sh script mkvfd creates a virtual FD containing a Linux root file system based on BusyBox. After booting up, the Linux kernel mounts the FD as the root file system and runs on the same FD. The screen of running such a single-FD Linux is similar to Figure 13.13, except that it does not load any ramdisk image during booting.

3.4. Hard Disk Booter

In this section, we shall develop an online booter for booting MTX and big Linux images (bzImages) from hard disk partitions. The HD booter consists of 5 files; a bs.s in assembly, a bc.c in C, which includes io.c, bootMtx.c for booting MTX and bootLinux.c for booting Linux. During booting, it displays the hard disk partitions and prompts for a

partition number to boot. If the partition type is MTX (90) or Linux (83), it allows the user to enter a filename to boot. If the user enters only the return key, it boots /boot/mtx or /boot/vmlinuz by default. When booting Linux it also supports an initial RAM disk image. For non-MTX/Linux partitions, it acts as a chain-booter to boot other operating systems, such as Windows. Since booting MTX is much simpler, we shall only discuss the Linux part of the HD booter.

The HD booter consists of five logical parts. Each part is essentially an independent programming task, which can be solved separately. For example, we may write a C program to display the partitions of a hard disk, and test it under Linux first. Similarly, we may write a program, which finds the inode of a file in an EXT2 file system and print its disk blocks. When the programs are tested to be working, we adapt them to the 16-bit environment as parts of the booter. The following describes the logical components of the HD booter.

3.4.1. I/O and Memory Access Functions

A HD booter is no longer limited to 512 or 1024 bytes. With a larger code size, we shall implement a set of I/O functions to provide better user interface during booting. Specifically, we shall implement a gets() function, which allows the user to input bootable image filename and boot parameters, and a printf() function for formatted printing. First, we show the gets() function.

```
#define MAXLEN 128
char *gets(char s[ ]) // caller must provide REAL memory s[MAXLEN]
{
    char c, *t = s; int len=0;
    while( (c=getc()) != '\r' && len < MAXLEN-1){
        *t++ = c; putc(c); len++;
    }
    *t = 0; return s;
}
```

For outputs, we first implement a printu() function, which prints unsigned short integers.

```
char *ctable = "0123456789ABCDEF";
u16 BASE = 10; // for decimal numbers
int rpu(u16 x)
{
    char c;
    if (x){
        c = ctable[x % BASE];
        rpu(x / BASE);
        putc(c);
    }
}
int printu(u16 x)
{
    (x==0)? putc('0') : rpu(x);
    putc(' ');
}
```

The function `rpu(x)` recursively generates the digits of `x % 10` in ASCII and prints them on the return path. For example, if `x=123`, the digits are generated in the order of '3', '2', '1', which are printed as '1', '2', '3' as they should. With `printu()`, writing a `printd()` to print signed short integers becomes trivial. By setting `BASE` to 16, we can print in hex. By changing the parameter type to `u32`, we can print long values, e.g. LBA disk sector and inode numbers. Assume that we have `prints()`, `printd()`, `printu()`, `printx()`, `printl()` and `printX()`, where `printl()` and `printX()` print 32-bit values in decimal and hex, respectively. Then write a `printf(char *fmt,...)` for formatted printing, where `fmt` is a format string containing conversion symbols `%c`, `%s`, `%u`, `%d`, `%x`, `%l`, `%X`.

```
int printf(char *fmt, ...) // some C compiler requires the three dots
{
    char *cp = fmt;          // cp points to the fmt string
    u16 *ip = (u16 *)&fmt + 1; // ip points to first item
    u32 *up;                  // for accessing long parameters on stack
    while (*cp){              // scan the format string
        if (*cp != '%'){      // spit out ordinary chars
            putc(*cp);
            if (*cp=='\n')     // for each '\n'
                putc('\r');    // print a '\r'
            cp++; continue;
        }
        cp++;                 // print item by %FORMAT symbol
        switch(*cp){
            case 'c' : putc(*ip); break;
            case 's' : prints(*ip); break;
            case 'u' : printu(*ip); break;
            case 'd' : printd(*ip); break;
            case 'x' : printx(*ip); break;
            case 'l' : printl(*(u32 *)ip++); break;
            case 'X' : printX(*(u32 *)ip++); break;
        }
        cp++; ip++;           // advance pointers
    }
}
```

The simple `printf()` function does not support field width or precision but it is adequate for the print task during booting. It would greatly improve the readability of the booter code. The same `printf()` function will also be used later in the MTX kernel. When booting a big Linux bzImage, the booter must get the number of `SETUP` sectors to determine how to load the various pieces of the image. After loading the image, it must set the boot parameters in the loaded `BOOT` and `SETUP` sectors for the Linux kernel to use. To do these, we implement the `get_byte()/put_byte()` functions in C, which are similar to the traditional `peek()/poke()` functions.

```
u8 get_byte(u16 segment, u16 offset)
{
    u8 byte;
    u16 ds = getds();        // getds() in assembly returns DS value
    setds(segment);          // set DS to segment
    byte = *(u8 *)offset;
    setds(ds);               // setds() in assembly restores DS
    return byte;
}
```

```

}
void put_byte(u8 byte, u16 segment, u16 offset)
{
    u16 ds = getds();    // save DS
    setds(segment);      // set DS to segment
    *(u8 *)offset = byte;
    setds(ds);           // restore DS
}

```

Similarly, we can implement `get_word()/put_word()` for reading/writing 2-byte words. These functions allow the booter to access memory outside of its own segment.

3.4.2. Read Hard Disk LBA Sectors

Unlike floppy disks, which use CHS addressing, large hard disks use Linear Block Addressing (LBA), in which disk sectors are accessed linearly by 32 or 48 bits sector numbers. To read hard disk sectors in LBA, we may use the extended BIOS INT13-42 (INT 0x13, AH=0x42) function. The parameters to INT13-42 are specified in a Disk Address Packet (DAP) structure.

```

struct dap{          // DAP structure for INT13-42
    u8  len;          // dap length=0x10 (16 bytes)
    u8  zero;         // must be 0
    u16 nsector;      // actually u8; sectors to read=1 to 127
    u16 addr;         // memory address = (segment, addr)
    u16 segment;      // segment value
    u32 sectorLo;     // low 4 bytes of LBA sector#
    u32 sectorHi;     // high 4 bytes of LBA sector#
};

```

To call INT13-42, we define a global `dap` structure and initialize it once, as in

```

struct dap dap, *dp=&dap; // dap and dp are globals in C
dp->len = 0x10;           // dap length = 0x10
dp->zero = 0;             // this field must be 0
dp->sectorHi = 0;         // assume 32-bit LBA, high 4-byte always 0
// other fields will be set when the dap is used in actual calls

```

Within the C code, we may set `dap`'s segment, then call `getSector()` to load one disk sector into the memory location (segment, offset), as in

```

int getSector(u32 sector, u16 offset)
{
    dp->nsector = 1;
    dp->addr    = offset;
    dp->sectorLo= sector;
    disk();
}

```

where `diskr()` is in assembly, which uses the global `dap` to call BIOS `int13-42`.

```

!----- assembly code -----
    .globl _diskr, _dap ! _dap is a global dap struct in C

```

```

_diskr:
    mov     dx, #0x0080    ! device=first hard drive
    mov     ax, #0x4200    ! aH=0x42
    mov     si, #_dap      ! (ES,SI) points to _dap
    int     0x13           ! call BIOS INT13-42 to read sectors
    jb      _error         ! to error() if CarryBit is set (read failed)
    ret

```

Similarly, the function

```

int getblk(u32 blk, u16 offset, u16 nblk)
{
    dp->nsectors = nblk*SECTORS_PER_BLOCK; // max value=127
    dp->addr      = offset;
    dp->sectorLo  = blk*SECTORS_PER_BLOCK;
    disk_r();
}

```

loads nblk contiguous disk blocks into memory, beginning from (segment, offset), where nblk <= 15 because dp->nsectors <= 127.

3.4.3. Boot Linux bzImage with Initial Ramdisk Image

When booting a Linux bzImage, the image's BOOT+SETUP are loaded to 0x9000 as before but the Linux kernel is loaded to the physical address 0x100000 (1MB) in high memory. If a RAM disk image is specified, it is also loaded to high memory. Since the PC is in 16-bit real mode during booting, it cannot access memory above 1MB directly. Although we may switch the PC to protected mode, access high memory and then switch back to real-mode afterwards, doing these requires a lot of work. A better way is to use BIOS INT15-87, which is designed to copy memory between real and protected modes. Parameters of INT15-87 are specified in a Global Descriptor Table (GDT).

```

struct GDT
{
    u32 zeros[4];          // 16 bytes 0's for BIOS to use
    // src address
    u16 src_seg_limit;     // 0xFFFF = 64KB
    u32 src_addr;          // low 3 bytes of src addr, high_byte=0x93
    u16 src_hiword;        // 0x93 and high byte of 32-bit src addr
    // dest address
    u16 dest_seg_limit;    // 0xFFFF = 64KB
    u32 dest_addr;         // low 3 bytes of dest addr, high_byte=0x93
    u16 dest_hiword;       // 0x93 and high byte of 32-bit dest addr
    // BIOS CS DS
    u32 bzeros[4];
};

```

The GDT specifies a src address and a dest address; both are 32-bit physical addresses. However, the bytes that form these addresses are not adjacent, which makes them hard to access. Although both src_addr and dest_addr are defined as u32, only the low 3 bytes are part of the address, the high byte is the access rights 0x93. Similarly, both src_hiword and dest_hiword are defined as u16 but only the high byte is the 4th address byte; the low

byte is again the access rights 0x93. As an example, if we want to copy from the real address 0x00010000 (64KB) to 0x01000000 (16MB), a GDT can be initialized as follows.

```
init_gdt(struct GDT *p)
{   int i;
    for (i=0; i<4; i++)
        p->zeros[i] = p->bzeros[i] = 0;
    p->src_seg_limit = p->dest_seg_limit = 0xFFFF; // 64KB segments
    p->src_addr      = 0x93010000;                // bytes 0x00 00 01 93
    p->dest_addr     = 0x93000000;                // bytes 0x00 00 00 93
    p->src_hiword    = 0x0093;                    // bytes 0x93 00
    p->dest_hiword   = 0x0193;                    // bytes 0x93 01
}
```

The following code segment copies 4096 bytes from 0x00010000 (64KB) in real mode memory to 0x01000000 (16MB) in high memory.

C code:

```
struct GDT gdt;           // define a gdt struct
init_gdt(&gdt);          // initialize gdt as shown above
cp2himem();               // assembly code that does the copying
```

Assembly code:

```
.globl _cp2himem, _gdt    ! _gdt is a global GDT from C
_cp2himem:
    mov cx, #2048         ! CX=number of 2-byte words to copy
    mov si, #_gdt         ! (ES, SI) point to GDT struct
    mov ax, #0x8700       ! aH=0x87
    int 0x15              ! call BIOS INT15-87
    jc _error
    ret
```

Based on these, we can load the blocks of an image file to high memory as follows.

- (1). load a disk block (4KB or 8 sectors) to segment 0x1000;
- (2). cp2himem();
- (3). gdt.vm_addr += 4096;
- (4). repeat (1)-(3) for next block, etc.

This can be used as the basic loading scheme of a booter. For fast loading, the hd-booter tries to load up to 15 contiguous blocks at a time. It is observed that most PCs actually support loading 16 contiguous blocks at a time. On these machines, the images can be loaded in 64KB chunks.

3.4.4. Hard Disk Partitions

The partition table of a hard disk is in the MBR sector at the byte offset 446 (0x1BE). The table has 4 entries, each defined by a 16-byte partition structure, which is

```
struct partition {
    u8  drive;           // 0x80 - active
    u8  head;            // starting head
    u8  sector;          // starting sector
```

```

    u8  cylinder;      // starting cylinder
    u8  sys_type;      // partition type
    u8  end_head;      // end head
    u8  end_sector;    // end sector
    u8  end_cylinder;  // end cylinder
    u32 start_sector;  // starting sector counting from 0
    u32 nr_sectors;    // number of sectors in partition
};

```

If a partition is EXTEND type (5), it can be divided into more partitions. Assume that partition P4 is EXTEND type and it is divided into extend partitions P5, P6, P7. The extend partitions form a link list, as shown in Figure 3.17.

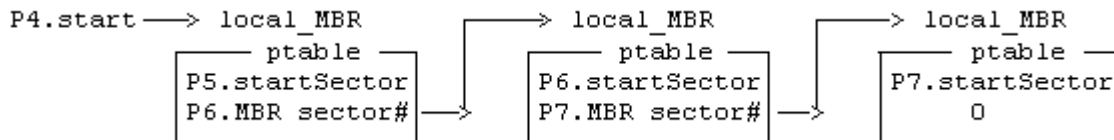


Figure 3.17. Link List of Extended Partitions

The first sector of each extend partition is a local MBR. Each local MBR has a partition table, which contains only two entries. The first entry defines the start sector number and size of the extend partition. The second entry points to the next local MBR. All the local MBR's sector numbers are relative to P4's start sector. As usual, the link list ends with a 0 in the last local MBR. In a partition table, the CHS values are valid only for disks smaller than 8GB. For disks larger than 8GB but fewer than 4G sectors, only the last 2 entries, start_sector and nr_sectors, are meaningful. Therefore, the booter should only display the type, start sector and size of the partitions.

3.4.5. Find and Load Linux Kernel and initrd Image Files

The steps used to find a Linux bzImage or RAM disk image are essentially the same as before. The main differences stem from the need to traverse large EXT2/EXT3 file systems on hard disks.

(1). In a hard disk partition, the superblock of an EXT2/EXT3 file system is at the byte offset 1024. A booter must read the superblock to get the values of s_first_data_block, s_log_block_size, s_inodes_per_group and s_inode_size, where s_log_block_size determines the block size, which in turn determines the values of group_desc_per_block, inodes_per_block, etc. These values are needed when traversing the file system.

(2). A large EXT2/EXT3 file system may have many groups. Group descriptors begin at the block (1+s_first_data_block), which is usually 1. Given a group number, we must find its group descriptor and use it to find the group's inodes start block.

(3). The central problem is how to convert an inode number to an inode. The following code segment illustrates the algorithm, which amounts to applying Mailman's algorithm twice.


```

/***** Algorithm: Convert inode number to inode *****/
(a). Compute group# and offset# in that group
    group    = (ino-1) / inodes_per_group;
    inumber  = (ino-1) % inodes_per_group;

(b). Find the group's group descriptor
    gdblck = group / desc_per_block;  // which block this GD is in
    gdisp  = group % desc_per_block;  // which GD in that block

(c). Compute inode's block# and offset in that group
    blk=inumber / inodes_per_block;  // blk# r.e.to group inode_table
    disp=inumber % inodes_per_block; // inode offset in that block

(d). Read group descriptor to get group's inode table start block#
    getblk(1+first_data_block+gdblck, buf, 1); // GD begin block
    gp = (GD *)buf + gdisp;  // it's this group desc.
    blk += gp->bg_inode_table; // blk is r.e. to group's inode_table
    getblk(blk, buf, 1);      // read the disk block containing inode
    INODE *ip=(INODE *)buf+(disp*iratio); //iratio=2 if inode_size=256

```

When the algorithm ends, INODE *ip should point to the file's inode in memory.

(4). Load Linux Kernel and Ramdisk Image to High Memory: With getblk() and cp2hmem(), loading kernel image to 1MB in high memory is straightforward. The only complication is when the kernel image does not begin at a block boundary. For example, if the number of SETUP sectors is 12, then 5 sectors of the kernel are in block1, which must be loaded to 0x100000 first before we can load the remaining kernel by blocks. In contrast, if the number of SETUP sectors is 23, then BOOT and SETUP are in the first 3 blocks and kernel begins at block #3. In this case, we can load the entire kernel by blocks without having to deal with fractions of a block at the beginning. Although the hd-booter handles these cases properly, it is certainly a pain. It would be much better if the Linux kernel of every bzImage begins at a block boundary. This can be done quite easily by modifying a few lines in the Linux tools program when it assembles the various pieces into a bzImage file. Why Linux people don't do that is beyond me.

Next, we consider loading RAM disk images. An excellent overview on Linux initial RAM disk (initrd) is in [Jones, 2006]. Slackware [Slackware Linux] also has an initrd HOWTO file. An initrd is a small file system, which is used by the Linux kernel as a temporary root file system when the kernel starts up. The initrd contains a minimal set of directories and executables, such as sh, the ismod tool and the needed driver modules. While running on initrd, the Linux kernel typically executes a sh script, initrc, to install the needed driver modules and activate the real root device. When the real root device is ready, the Linux kernel abandons the initrd and mounts the real root file system to complete a 2-stage boot up process. The reason of using an initrd is as follows. During booting, Linux's startup code only activates a few standard devices, such as FD and IDE/SCSI HD, as possible root devices. Other device drivers are either installed later as modules or not activated at all. This is true even if all the device drivers are built into the Linux kernel. Although it is possible to activate the needed root device by altering the kernel's startup code, the question is, with so many different Linux system configurations,

which device to activate? An obvious answer is to activate them all. Such a Linux kernel would be humongous in size and rather slow to boot up. For example, in some Linux distribution packages the kernel images are larger than 4MB. An initrd image can be tailor-built with instructions to install only the needed driver modules. This allows a single generic Linux kernel to be used in all kinds of Linux system configurations. In theory, a generic Linux kernel only needs the RAM disk driver to start. All other drivers may be installed as modules from the initrd. There are many tools to create an initrd image. A good example is the mkinitrd command in Linux. It creates an initrd.gz file and also an initrd-tree directory containing the initrd file system. If needed, the initrd-tree can be modified to generate a new initrd image. Older initrd.gz images are compressed EXT2 file systems, which can be uncompressed and mounted as a loop file system. Newer initrd images are cpio archive files, which can be manipulated by the cpio utility program. Assume that initrd.img is a RAM disk image file. First, rename it as initrd.gz and run gunzip to uncompress it. Then run

```
mkdir temp; cd temp; # use a temp DIR
cpio -id < ../initrd # extract initrd contents
```

to extract the contents. After examining and modifying files in initrd-tree, run

```
find . | cpio -o -H newc | gzip > ../initrd.gz
```

to create a new initrd.gz file.

Loading initrd image is similar to loading kernel image, only simpler. There is no specific requirement on the loading address of initrd, except for a maximum high address limit of 0xFE000000. (The reader may consult Chapter 15 on SMP for reasons). Other than this restriction, any reasonable loading address seems to work fine. The hd-booter loads the Linux kernel to 1MB and initrd to 32MB. After loading completes, the booter must write the loading address and size of the initrd image to SETUP at the byte offsets 24 and 28, respectively. Then it jumps to execute SETUP at 0x9020. Early SETUP code does not care about the segment register settings. In kernel 2.6, SETUP requires DS=0x9000 in order to access BOOT as the beginning of its data segment.

3.4.6. Linux and MTX Hard Disk Booter

A complete listing of the hd-booter code is in BOOTERS/HD/MBR.ext4/. The booter can boot both MTX and Linux with initial RAM disk support. It can also boot Windows by chain-booting. For the sake of brevity, we only show the booting Linux part here.

```
!----- hd-booter's bs.s file -----
    BOOSEG = 0x9800
    SSP    = 32*1024      ! 32KB bss + stack; may be adjusted
    .globl _main, _prints, _dap, _dp, _bsector, _vm_gdt    ! IMPORT
    .globl _diskr, _getc, _putc, _getds, _setds,          ! EXPORT
    .globl _cp2himem, _jmp_setup
! MBR loaded at 0x07C0. Load entire booter to 0x9800
start: mov  ax, #BOOTSEG
      mov  es, ax
      xor  bx, bx      ! clear BX = 0
      mov  dx, #0x0080 ! head 0, HD
      xor  cx, cx
      incb cl          ! cyl 0, sector 1
```

```

        incb cl
        mov ax, #0x0220      ! READ 32 sectors, booter size up to 16KB
        int 0x13
! far jump to (0x9800, next) to continue execution there
        jmp next, BOOSEG      ! CS=BOOTSEG, IP=next
next:
        mov ax, cs           ! set CPU segment registers
        mov ds, ax           ! we know ES,CS=0x9800. Let DS=CS
        mov ss, ax
        mov es, ax           ! CS=DS=SS=ES=0x9800
        mov sp, #SSP         ! 32 KB stack
        call _main           ! call main() in C
        test ax, ax          ! check return value from main()
        je error             ! main() return 0 if error
        jmp 0x7C00,0x0000     ! otherwise, as a chain booter

_diskr:
        mov dx, #0x0080      ! drive=0x80 for HD
        mov ax, #0x4200
        mov si, #_dap
        int 0x13             ! call BIOS INT13-42 read the block
        jb error             ! to error if CarryBit is on
        ret

error:
        mov bx, #bad
        push bx
        call _prints
        int 0x19             ! reboot
bad:    .asciz "\n\rError!\n\r"
_jmp_setup:
        mov ax, 0x9000       ! for SETUP in 2.6 kernel:
        mov ds, ax           ! DS must point at 0x9000
        jmp 0, 0x9020        ! jmp to execute SETUP at 0x9020

_getc:  ! same as before
_putc:  ! same as before
_getds: ! return DS value
_setds: ! set DS to a segment
!----- cp2himem() -----
! for each batch of k<=16 blocks, load to RM=0x10000 (at most 64KB)
! then call cp2himem() to copy it to      VM=0x100000 + k*4096
!-----
_cp2himem:
        push bp
        mov bp, sp
        mov cx, 4[bp]        ! words to copy (32*1024 or less)
        mov si, #_vm_gdt
        mov ax, #0x8700
        int 0x15
        jc error
        pop bp
        ret

/***** Algorithm of hd-booter's bc.c file *****/
#define BOOTSEG 0x9800
#include "bio.c"           // I/O functions such as printf()
#include "bootLinux.c"     // C code of Linux booter
int main()
{

```

```

(1). initialize dap for INT13-42 calls;
(2). read MBR sector;
(3). print partition table;
(4). prompt for a partition to boot;
(5). if (partition type == LINUX)
    bootLinux(partition);    // no return
(6). load partition's local MBR to 0x07C0;
    chain-boot from partition's local MBR;
}
/***** Algorithm of bootLinux.c file *****/
boot-Linux-bzImage Algorithm:
{
(1). read superblock to get blockSize,inodeSize,inodes_per_group
(2). read Group Descriptor 0 to get inode start block
(3). read in the root INODE and let INODE *ip point at root INODE
(4). prompt for a Linux kernel image filename to boot
(5). tokenize image filename and search for image's INODE
(6). handle symbolic-link filenames
(7). load BOOT+SETUP of Linux bzImage to 0x9000;
(8). set video mode word at 506 in BOOT to 773 (for small font).
(9). set root dev word at 508 in BOOT to (0x03, pno) (/dev/hdapno)
(10). set bootflags word at offset 16 in SETUP to 0x2001
(11). compute number of kernel sectors in last block of SETUP
(12). load kernel sectors to 0x1000, then cp2himem() to 1MB
(13). load kernel blocks to high memory, each time load 64KB
(14). load initrd image to 32 MB in high memory
(15). write initrd address and size to offsets (24,28) in SETUP
(16). jmp_setup() to execute SETUP code at 0x9020
}

```

In the above algorithm, step (8) is optional. Step (9) sets the root device, which is needed only if no initrd image is loaded. With an initrd image, the root device is determined by the initrd image. Step (10) is mandatory, which tells SETUP that the kernel image is loaded by an "up-to-date" boot loader. Otherwise, the SETUP code would consider the loaded kernel image invalid and refuse to start up the Linux kernel.

3.4.7. Boot EXT4 Partitions

At the time of this writing, many Linux distributions are switching to EXT4 [Cao et al., 2007] as the default file system. It is fairly easy to modify the booter to boot MTX and Linux from EXT4 partitions. Here, we briefly describe the EXT4 file system and the needed modifications to the HD booter.

(1). In EXT4, the `i_block[15]` array of an inode contains a header and 4 extents structures, each 12 bytes long, as shown below.

```

|<----- u32 i_block[15] area ----->|
|header|extent1|extent2|extent3|extent4|
struct ext3_extent_header {
    u16  eh_magic;        // 0xF30A
    u16  eh_entries;      // number of valid entries
    u16  eh_max;          // capacity of store in entries
    u16  eh_depth;        // has tree real underlying blocks?

```

```

        u32 eh_generation; // generation of the tree
    };

    struct ext3_extent {
        u32 ee_block;      // first logical block extent covers
        u16 ee_len;        // number of blocks covered by extent
        u16 ee_start_hi;   // high 16 bits of physical block
        u32 ee_start;      // low 32 bits of physical block
    };

```

The root directory does not use extents, so `i_block[0]` is still the first data block.

(2). The GD and INODE types are the same as they are in EXT2, but the INODE size is 256 bytes. The SUPER block's magic number is also the same as in EXT2, but we may test `s_feature_incompat (> 0x240)` to determine whether it's an EXT4 file system.

(3). Blocks in each extent are contiguous. There is no need to scan for contiguous blocks when loading an image; just load a sequence of blocks directly. For HDs, the block size is 4KB. The maximum number of blocks per loading is still limited to 16 or less. Shown below are the `search()` and `load()` functions for EXT4 file system. Integrating them into the HD booter is left as an exercise.

```

/***** serach for name in an EXT4 DIR INODE *****/
u32 search(INODE *ip, char *name)
{
    u16 i; u32 ino;
    struct ext3_extent_header *hdp;
    struct ext3_extent *ep;
    char buf[BLK];
    hdp = (struct ext3_extent_header *)&(ip->i_block[0]);
    ep = (struct ext3_extent *)&(ip->i_block[3]);
    for (i=0; i<4; i++){
        if (hdp->eh_entries == 0){
            getblk((u32)ip->i_block[0], buf, 1);
            i = 4; // no other extents
        }
        else{
            ep = (struct ext3_extent *)&(ip->i_block[3]);
            getblk((u32)ep->ee_start, buf, 1);
        }
        if (ino = find(buf, name)) // find name string in buf[ ]
            return ino;
    }
    return 0;
}

/***** load blocks of an INODE with EXT4 extent *****/
int loadExt4(INODE *ip, u16 startblk)
{
    int i,j,k,remain; u32 *up;
    struct ext3_extent_header *hdp;
    struct ext3_extent *ep;
    int ext;
    u32 fblk, beginblk;
    hdp = (struct ext3_extent_header *)ip->i_block;
    ep = (struct ext3_extent *)ip->i_block + 1;

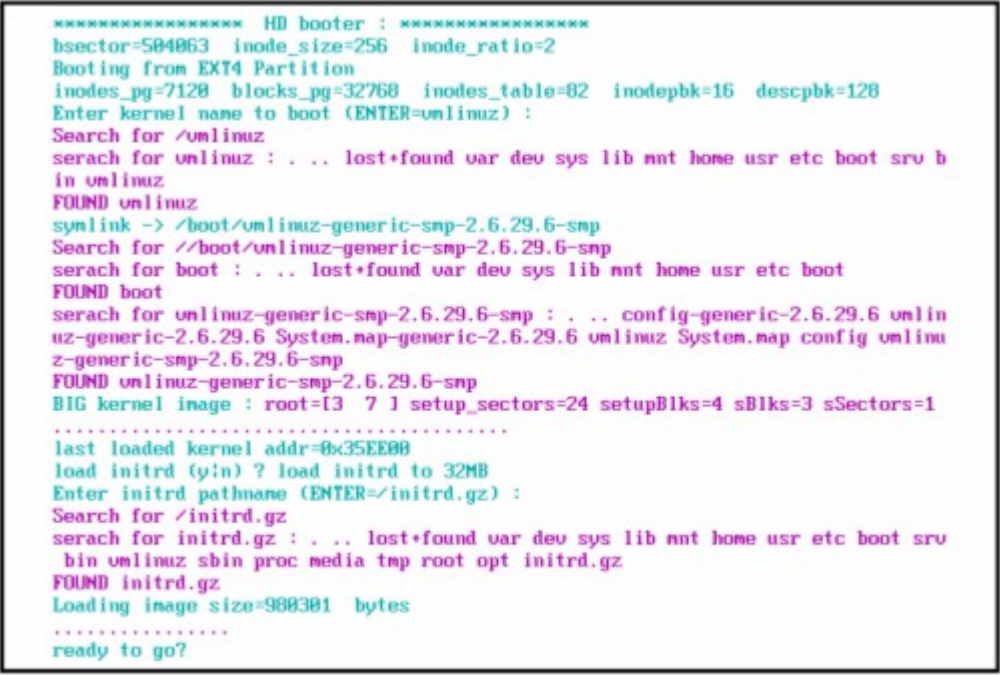
```

```

ext = 1;
while(1){
    if (ep->ee_len==0)
        break;
    beginblk = 0;
    if (ext==1) // if first extent: begin from startblk
        beginblk = startblk;
    k = 16; // load 16 continuous blocks at a time
    fblk = ep->ee_start + beginblk;
    remain = ep->ee_len - beginblk;
    while(remain >= k){
        getblk((u32)(fblk), 0, k);
        cp_vm(k, '.');
        fblk += k;
        remain -= k;
    }
    if (remain){
        getblk((u32)(fblk), 0, remain);
        cp_vm(remain, '.');
    }
    ext++; ep++; // next extent
}
}

```

Figure 3.18 shows the screen of the hd-booter when booting a generic Linux kernel with initial RAM disk image, `initrd.gz`, from an EXT4 partition.



```

***** HD booter : *****
bsector=504063 inode_size=256 inode_ratio=2
Booting from EXT4 Partition
inodes_pg=7120 blocks_pg=32768 inodes_table=02 inodespbk=16 descpbk=120
Enter kernel name to boot (ENTER=vmlinuz) :
Search for /vmlinuz
serach for vmlinuz : . . . lost+found var dev sys lib mnt home usr etc boot srv b
in vmlinuz
FOUND vmlinuz
symlink -> /boot/vmlinuz-generic-smp-2.6.29.6-smp
Search for //boot/vmlinuz-generic-smp-2.6.29.6-smp
serach for boot : . . . lost+found var dev sys lib mnt home usr etc boot
FOUND boot
serach for vmlinuz-generic-smp-2.6.29.6-smp : . . . config-generic-2.6.29.6 vmlin
uz-generic-2.6.29.6 System.map-generic-2.6.29.6 vmlinuz System.map config vmlinu
z-generic-smp-2.6.29.6-smp
FOUND vmlinuz-generic-smp-2.6.29.6-smp
BIG kernel image : root=[3 7 1] setup_sectors=24 setupBlks=4 sBlks=3 sSectors=1
.....
last loaded kernel addr=0x35EE00
load initrd (y;n) ? load initrd to 32MB
Enter initrd pathname (ENTER=/initrd.gz) :
Search for /initrd.gz
serach for initrd.gz : . . . lost+found var dev sys lib mnt home usr etc boot srv
bin vmlinuz/sbin/proc/media/tmp/root/opt/initrd.gz
FOUND initrd.gz
Loading image size=988381 bytes
.....
ready to go?

```

Figure 3.18. Booting Linux bzImage with initrd from EXT4 Partition

3.4.8. Install HD Booter

Now that we have a hard disk booter, the next problem is where to install it? Obviously, the beginning part of the booter must be installed in the HD's MBR since that's where the booting process begins. The question is where to install the remaining parts of the booter? The location chosen must not interfere with the hard disk's normal contents. At the same time it must be easy for the stage1 booter to find. The question has an interesting answer. By convention, each HD partition begins at a (logical) track boundary. Since the MBR is already in track 0, partition 1 actually begins from track 1. A track usually has 63 sectors. We can certainly put a fairly big and powerful booter in the unused space of track 0. Unfortunately, once the good news gets around, it seems that everybody tries to use that hidden space for some special usage. For example, GRUB installs its stage2 booters there, so does our hd-booter. Naturally, as a Chinese proverb says, "A single mountain cannot accommodate two tigers", only one tiger can live there at a time. The hd-booter can be installed to a HD as follows.

```
(1) dd if=hd-booter of=/dev/hda bs=16 count=27
(2) dd if=hd-booter of=/dev/hda bs=512 seek=1
```

Assume that the booter size is less than 31KB (the hd-booter size is about 10KB). Step (1) dumps the first 432 bytes of the booter to the MBR without disturbing the partition table, which begins at byte 444. Step (2) dumps the entire booter to sectors 1 and beyond. During booting, BIOS loads the MBR to 0x07C00 and executes the beginning part of the hd-booter. The hd-booter reloads the entire booter, beginning from sector 1, to 0x98000 and continues execution in the new segment. The actual number of sectors to load can be adjusted to suit the booter size, but loading a few extra sectors causes no harm.

Although installing the hd-booter is simple, a word of caution is in order. Murphy's law says anything that can go wrong will go wrong. Writing to a hard disk's MBR is very risky. A simple careless mistake may destroy the partition table and/or corrupt the HD contents, rendering the HD either non-bootable or useless. It is therefore advised not to install the booter to a HD unless you are absolutely sure of what you are doing. Before attempting to install the booter, it's a good idea to write down the HD's partition table on a piece of paper in case you have to restore it. A safer way to test the HD booter is to install it to a floppy disk. For FD drives that support loading cylinders, we only need to modify one line in the above assembly code: change `mov dx, #0x0080` to `mov dx, #0x0000`, so that the booter will be re-loaded from a FD when it begins to run. Once the booter starts running, it actually boots from the HD. Since the HD is accessed in read-only mode, the scheme should be safe. Instead of a real HD, the reader may use a virtual HD. Similarly, the HD booter may also be installed to a USB drive. In that case, no changes are needed.

3.5. CD/DVD-ROM Booter

A bootable CD/DVD is created in two steps. First, create an iso9660 file system [Standard ECMA-119, 1987] containing a CD/DVD booter. Then write the iso image to a CD/DVD by a CD/DVD burning tool. The resulting CD/DVD is bootable. If desired, the iso file can also be used directly as a virtual CD. In this section, we shall show how to develop booter programs for CD/DVD booting.

3.5.1. Emulation CDROM Booting

From a programming point of view, emulation booting is trivial. There is not much one needs to (or can) do other than preparing a bootable disk image. The following shows how to do emulation booting.

3.5.1.1. Emulation-FD Booting

Assume that fdimage is a bootable floppy disk image (size = 1.44MB). Under Linux, use the sh command

```
mkisofs -o /tmp/fcd.iso -v -d -R -J -N -b fdimage -c boot.catalog ./
```

to create a /tmp/fcd.iso file from the current directory. The reader may consult Linux man page of mkisofs for the meaning of the various flags. The iso file is a bootable CD image. It can be written to a real CD/DVD disc by using a suitable CD/DVD burning tool, such as Nero or K3b under Linux. It can also be used as a virtual CD on most virtual machines. Then boot from either a real or a virtual CD/DVD. After booting up, the environment is exactly the same as that of booting from a floppy disk.

Example 1: BOOTERS/CD/emuFD demonstrates emulation-FD booting. It contains a MTX system, MTXimage, based on MTX5.1 of Chapter 5. When creating a bootable CD image, it is used as the emulation-boot image. Upon booting up from the CD, MTX runs as if it had been booted up from a FD. As pointed out before, the MTX kernel can only access the MTXimage on the CD as if it were a FD drive, but it cannot access anything else on the CD.

3.5.1.2. Emulation-HD Booting

Similarly, assume that hdimage is a single-partition hard disk image with a HD booter installed in the MBR. Under Linux, use the sh command

```
mkisofs -o /tmp/hcd.iso -v -d -R -J -N -b hdimage -hard-disk-booting -c boot.catalog .
```

to create a bootable CD image and burn the hcd.iso file to a CD/DVD disc. After booting up, the environment is exactly the same as that of booting from the first hard disk.

Example 2: BOOTER/CD/emuHD demonstrates emulation-HD booting. In the emuHD directory, hdimage is single-partition hard disk image. It contains a MTX system in partition 1 and a MTX booter (hd-booter of Section 3.4.5) in MBR. In the example, the hdimage is used as the hard-disk-boot image to create a bootable CDROM image. When booting from the CDROM, the sequence of actions is identical to that of booting MTX from a HD partition. When the MTX kernel runs, the environment is the same as that of running from the C: drive. All I/O operations to the emulated hard disk use INT13-42 calls to BIOS. Again, the MTX kernel can only access the hdimage but nothing else on the CDROM.

3.5.2. No-emulation CDROM Booting

In no-emulation booting, if the loading requirements are simple, e.g. just load the OS image to a segment in real-mode memory, then there is no need for a separate booter because the entire OS image can be loaded by BIOS during booting.

3.5.2.1. No-emulation Booting of MTX

Example 3: BOOTERS/CD/MTXCD demonstrates no-emulation booting of MTX. It contains a MTX system, which is again based on MTX5.1. However, the MTX kernel is modified to include an iso loader and a simple iso file system traversing program. The MTX kernel is used as the no-emulation booting image. During booting, the entire MTX kernel is loaded to the segment 0x1000 and runs from there. When the MTX starts to run, it must create a process with a user mode image from a /bin/u1 file, which means it must be able to read the CDROM contents. Loading the user mode image file is done by the isoloader. The program cd.c supports basic iso9660 file system operations, such as ls, cd, pwd and cat. These allow a process to navigate the file system tree on the CDROM. The example is intended to show that a booted up OS kernel can access the CDROM contents if it has drivers to interpret the iso file system on the CDROM.

3.5.2.2. No-emulation Linux Booter

In no-emulation CDROM booting, a separate booter is needed only if the loading requirements of an OS image are non-trivial, such as that of Linux. In the following, we shall develop an iso-booter for booting Linux bzImage with initial RAM disk support from CDROM. To do this, we need some background information about the iso9660 file system, which are summarized below.

For data storage, CDROM uses 2048-byte sectors, which are addressed in LBA just like HD sectors. The data format in an iso9660 file system represents what may be called a masterpiece of legislative compromise. It supports both the old 8.3 filenames of DOS and, with Rock Ridge extension, it also supports Unix-style filenames and attributes. To accommodate machines using different byte orders, all multi-byte values are stored twice in both little-endian and big-endian formats. To support international encoding, chars in Joliet extension are stored in 16-bit Unicode. An iso9660 CDROM contains a sequence of Volume Descriptors (VDs), which begin at sector 16. Each VD has a type identifier, where 0=BOOT, 1=Primary, 2=Supplementary and 255=End of the VD table. Unix-style files are under the supplementary VD, which contains, among other thing, the following fields.

```
u8  type           = VD's type
u32 type_1_path_table = start sector of Little_endian path_table
u32 path_table_size  = path_table size in bytes.
root_directory_record = root DIR iso_directory_record
```

The steps of traversing a Unix-style file system on a CDROM are as follows.

1. From sector 16, read in and step through the Volume Descriptors to search for the Supplementary VD (SVD), which has type=2.
2. SVD.root_directory_record is an iso_directory_record (DIR) of 34 bytes.

```
struct iso_directory_record {
    unsigned char length;
```

```

    unsigned char ext_attr_length;
    char extent[8];

    char size[8];

    char date[7];

    unsigned char flags;
    char file_unit_size;
    char interleave;
    char volume_sequence_number[4];
    unsigned char name_len;

    char name[0];

};

```

3. Multi-byte values are stored in both little-endian and big-endian format. For example, `DIR.extent = char extent[8] = [4-byte-little-endian, 4-byte-big-endian]`. To get a DIR's extent (start sector), we may use `u32 extent = *(u32 *)DIR.extent`, which extracts only the first 4 little-endian bytes. Similarly for `DIR.size`, etc.

4. In an iso file system, FILE and DIR records are identical. Therefore, entries in a directory record are also directory records. The following algorithm shows how to search for a name string in a DIR record.

```

/** Algorithm of search for fname string in DIR */
DIR *search(DIR, fname)
{
    sector = DIR.extent (begin sector# of DIR record);
    while(DIR.size){
        read sector into a char buf[2048];
        char *cp = buf; DIR *dp = buf; // both point at buf beginning
        while(cp < buf+2048){
            each record has a length, a name_len and a name in 16-bit
            Unicode. Convert name to ascii, then compare with fname;
            if (found) we actually have fname's RECORD;
                return DIR record (pointer);
            else advance cp by record length, pull dp to next record;
        } // until buf[ ] end
        DIR.size -= 2048; sector++;
    } // until DIR.size=0
}

```

5. To search for the DIR record of a pathname, e.g. `/a/b/c/d`, tokenize the pathname into component name strings. Start from the root DIR, search for each component name in the

current DIR. The steps are similar to that of finding the inode of a pathname in an EXT2/EXT3 file system.

6. If we allow .. in a pathname, we must be able to get the parent of the current DIR. Similar to a Unix directory, the second entry in an iso9660 directory contains the extent of the parent directory. For each .. entry we may either return the parent DIR's extent or a DIR pointer to the second record. Alternatively, we may also search the path table to find the parent DIR's extent. This method is left as an exercise.

With this background information, we are ready to show the details of an iso-booter. First, the iso.h file contains the types of volume descriptor, directory record and path table. All entries are defined as char arrays. For ease of reference, arrays of size 1 are redefined simply as char. The primary and supplementary volume descriptors differ in only in 2 fields, flags and escape, which are unspecified in the former but specified in the latter. Since these fields are irrelevant during booting, we only use the supplementary volume descriptor. Both iso_directory_record and iso_path_table are open-ended structures, in which the name field may vary, depending on the name_len. When stepping through these records we must advance by the actual record length. Similarly, when copying a directory record we must use memcpy(p1,p2, p2->length) to ensure that the entire record is copied.

In the iso-booter, BOOTSEG is set to 0x07C0, rather than 0x9800. This is because many older PCs, e.g. some Dell and IBM Thinkpad laptops, seem to ignore the -boot-load-seg option and always load the boot image to the segment 0x07C0. For maximum compatibility, the iso-booter is loaded to the segment 0x07C0 and runs from there without relocation. This works out fine for Linux, which does not use the memory area between 0x07C0 and 0x1000. The iso-booter's bs.s file is the same as that of the hd-booter, with only a minor difference in the beginning part. When the iso-booter starts, it is already completely loaded in and it does not need to relocate. However, it must use the boot drive number passed in by BIOS, as shown below.

```
!----- iso-booter's bs.s file -----
! In no-emulation booting, many PCs always load booted image to 0x07C0.
! Only some PCs honor the -boot-load-seg=SEGMENT option. So use 0x07C0
!-----
        BOOTSEG  = 0x07C0
        SSP      = 32*1024
!       .globls : SAME as in hd-booter
        .globl   _drive      ! boot drive# in C code, passed in DL
        jmp     start,BOOTSEG ! upon entry, set CS to BOOTSEG=0x07C0
start:
        mov     ax,cs        ! set other CPU segment registers
        mov     ds,ax        ! we know ES,CS=0x07C0. Let DS=CS
        mov     ss,ax        ! SS = CS ==> all point to 0x07C0
        mov     es,ax
        mov     sp,#SSP      ! SP = 32KB
        mov     _drive,dx    ! save drive# from BIOS to _drive in C
        call    _main        ! call main() in C
! Remaining .s code: SAME AS in hd-booter but use the boot drive#
```

The iso-booter's C code and algorithms are also similar to the hd-booter. For the sake of brevity, we only show the parts that are unique to the iso-booter. A complete listing of the iso-booter code is in BOOTERS/CD/isobooter/ directory.

```

/*****iso-booter's bc.c file *****/
#define BOOTSEG 0x07C0
#include "iso.h"          // iso9660 file types
#include "bio.c"          // contains I/O functions
main()
{
    (1). initialize dap and vm_gdt for BIOS calls
    (2). find supplement Volume Descriptor to get root_dir_record
    (3). get linux bzImage filename to boot or use default=/vmlinuz;
    (4). load(filename);
    (5). loadrd("initrd.gz");
    (6). jmp_setup();
}
}
/***** iso-booter's bootLinux.c file *****/
u32 bsector;             // getnsector() base sector
u32 zsector, zsize;      // bzImage's begin sector# & size
struct vmgdt {            // same as in hd-booter }
init_vm_gdt(){            // same as in hd-booter }
// get nsectors from bsector+rsector to dp-segment
u16 getnsector(u16 rsector, u16 nsector);
{
    dp->nsector = nsector;
    dp->addr = (u16)0;      // load to dp->segment:0
    dp->s1 = (u32)(bsector + (u32)rsector); // rsector = offset
    readcd();              // same as diskr() but use boot drive#
}
// loadimage() : load 32 CD-sectors to high memory
int loadimage(u16 imageStart, u32 imageSize)
{
    u16 i, nsectors;
    nsectors = imageSize/2048 + 1;
    i = imageStart;
    // load 32 CD sectors at a time to 0x1000; then cp2himem();
    while(i < nsectors){
        getnsector(i, 32);
        cp2himem(32*1024);
        gdt->vm_addr += (u32)0x10000;
        putc('.');
        i += 32;
    }
}
// dirname() : convert DIR name in Unicode-2 to ascii in temp[ ]
char temp[256];
char *dirname(struct iso_directory_record *dirp)
{ int i;
  for (i=0; i<dirp->name_len; i+=2){
    temp[i/2] = dirp->name[i+1];
  }
  temp[dirp->name_len/2] = 0;
  return temp;
}

```

```

// search DIR record for name; return pointer to name's record
struct iso_directory_record *search(struct iso_directory_record *dirp,
char *name)
{
    char *cp, dname[256];
    int i, loop, count;
    u32 extent, size;
    struct iso_directory_record *ddp, *parent;
    printf("search for %s\n", name);
    extent = *(u32 *)dirp->extent;
    size = *(long*)dirp->size;
    loop = 0;
    while(size){
        count = 0;
        getSector(extent, rbuf);
        cp = rbuf;
        ddp = (struct iso_directory_record *)rbuf;
        if (strcmp(name, "..")==0){ // for .., return 2nd record pointer
            cp += ddp->length;
            ddp = (struct iso_directory_record *)cp;
            return ddp;
        }
        while (cp < rbuf + SECSIZE){
            if (ddp->length==0)
                break;
            strcpy(dname, dirname(ddp)); // assume supplementary VD only
            if (loop==0){ // . and .. only in the first sector
                if (count==0) strcpy(dname, ".");
                if (count==1) strcpy(dname, "..");
            }
            printf("%s ", dname);
            if (strcasecmp(dname, name)==0){ // ignore case
                printf(" ==> found %s : ", name);
                return ddp;
            }
            count++;
            cp += ddp->length;
            ddp = (struct iso_directory_record *)cp;
        }
        size -= SECSIZE;
        extent++;
        loop++;
    }
    return 0;
}

// getfile() : return pointer to filename's iso_record
struct iso_directory_record *getfile(char *filename)
{
    int i;
    struct iso_directory_record *dirp;
    tokenize(filename); // same as in hd-booter;
    dirp = root;
    for (i=0; i<nnames; i++){
        dirp = search(dirp, name[i]);
        if (dirp == 0){
            printf("no such name %s\n", name[i]);
            return 0;
        }
    }
}

```

```

        // check DIR type
        if (i < nnames-1){          // check DIR type but ignore symlinks
            if ((dirp->flags & 0x02) == 0){
                printf("%s is not a DIR\n", name[i]);
                return 0;
            }
        }
    }
}
return dirp;
}

int load_rd(char *rdname) // load_rd() : load initrd.gz image
{
    u32 rdstart, rdsize;           // initrd's start sector & size
    // (1). set vm_addr to initrd's loading address at 32MB
    dirp = getfile(rdname);
    rdstart = *(u32 *)dirp->extent; // start sector of zImage on CD
    rdsize = *(long *)dirp->size;   // size in bytes
    // (2). load initrd image
    dp->segment = 0x1000;
    bsector = rdstart;
    loadimage((u16)0, (u32)rdsize);
    // (3). write initrd loading address and size to SETUP
}

int load(char *filename) // load() : load Linux bzImage
{
    struct iso_directory_record *dirp;
    dirp = getfile(filename);
    // dirp now points at bzImage's RECORD
    zsector = *(u32 *)dirp->extent; // start sector of bzImage on CD
    zsize = *(long *)dirp->size;   // size in bytes
    /***** SAME AS in hd-booter except CD-sector size=2KB *****/
    get number of 512-byte setup sectors in filename's BOOT sector
    load BOOT+SETUP to 0x9000
    set boot parameters in loaded BOOT+SETUP
    load kernel fraction sectors in SETUP to 1MB in high memory
    *****/
    // continue to load kernel 2KB CD-sectors to high memory
    dp->segment = 0x1000;
    loadimage((u16)setupBlks, (u32)zsize);
    load_rd("/initrd.gz"); // assume initrd.gz filename
    jmp_setup();
}

```

Under Linux, run `mk` to generate a boot image file `iso-booter` as before. Next, run

```

mkisofs -o /tmp/iso-booter.iso -v -d -R -J -N -no-emul-boot \
        -boot-load-size 20 \ # (512-byte) sectors to load
        -b iso-booter \     # boot image file
        -c boot.catalog ./   # from files in current directory

```

to generate an iso image, which can be burned to a CD disc or used as a virtual CD.

3.5.3. Comparison with isolinux CDROM Booter

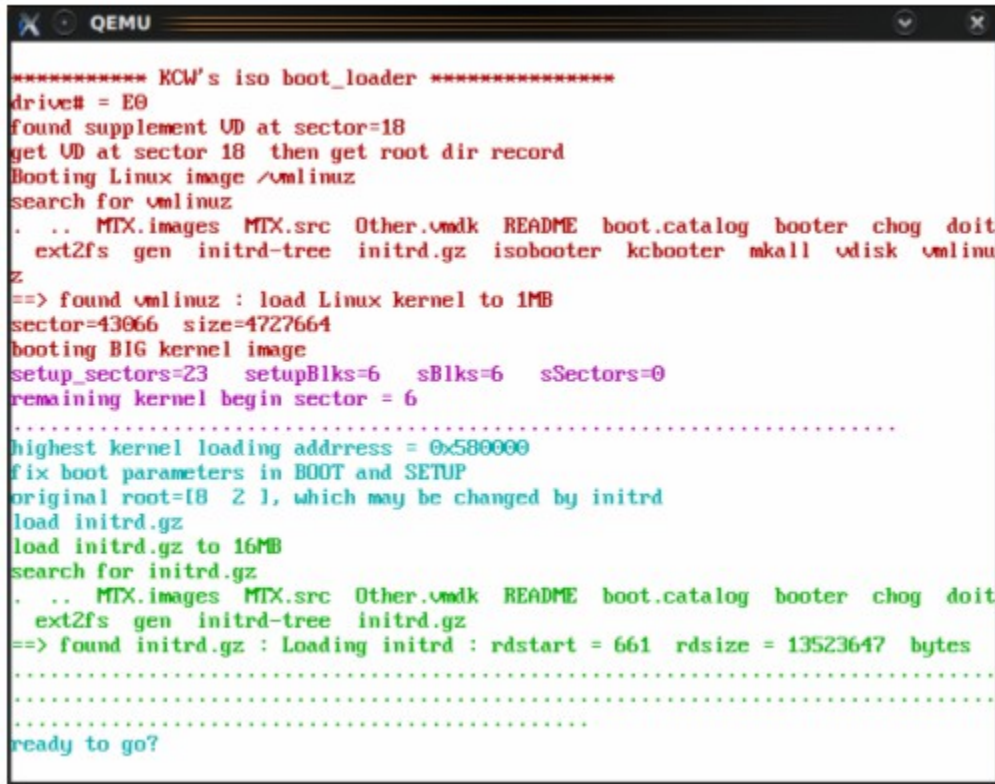
isolinux [Sylinux project] is a CD/DVD Linux boot-loader. It is used in almost all CD/DVD based Linux distributions. As an example, the bootable CD/DVD of Slackware Linux 13.1 distribution contains

```
|-- isolinux/  : isolinux.bin, isolinux.cfg, initrd.img
/-- |-- kernels/    : huge.s/bzImage: bootable Linux bzImage file
|-- slackware/ : Linux distribution packages
```

where isolinux.bin is the (no-emulation) booter of the CD/DVD. During booting, isolinux.bin consults isolinux.cfg to decide which Linux kernel to load. Bootable Linux kernels are in the kernels/ directory. The user may choose a kernel that closely matches the target system hardware or use the default kernel. With a kernel file name, isolinux.bin loads the kernel and the initial ramdisk image, initrd.img, which is compressed root file system based on BusyBox. Then it executes SETUP. When the Linux kernel starts, it mounts initrd.img as the root device.

Example 4. Replace isolinux booter with iso-booter: The iso-booter can be used to replace the isolinux booter in a Linux distribution. As a specific example, BOOTERS/CD/slackCD/ is a copy of the Slackware 13.1 boot CD but without the installing packages of Linux. It uses the iso-booter of this book to generate a bootable iso image. During booting, enter /kernels/bzImage as the kernel and /isolinux/initrd.img as the initial ramdisk image. Slackware's install environment should start up.

Example 5. Boot generic Linux bzImage with initrd.gz: In the BOOTERS/CD/linuxCD/ directory, vmlinuz is a generic Linux kernel, which must be booted with an initial ramdisk image. The initrd.gz file is generated by the mkinitrd command using files in the /boot/initrd-tree/ directory. The iso-booter can boot up a generic Linux kernel and load the initrd.gz for the Linux kernel to start. Figure 4.19 shows the booting screen of the iso-booter. It loads the Linux kernel to 1MB and initrd.gz to 16MB.



```
***** KCW's iso boot_loader *****
drive# = E0
found supplement UD at sector=18
get UD at sector 18 then get root dir record
Booting Linux image /vmlinuz
search for vmlinuz
. .. MTX.images MTX.src Other.vmdk README boot.catalog booter chog doit
  ext2fs gen initrd-tree initrd.gz isobooter kcbooter mkall wdisk vmlinu
Z
==> found vmlinuz : load Linux kernel to 1MB
sector=43066 size=4727664
booting BIG kernel image
setup_sectors=23 setupBlks=6 sBlks=6 sSectors=0
remaining kernel begin sector = 6
.....
highest kernel loading address = 0x580000
fix boot parameters in BOOT and SETUP
original root=[8 2 ], which may be changed by initrd
load initrd.gz
load initrd.gz to 16MB
search for initrd.gz
. .. MTX.images MTX.src Other.vmdk README boot.catalog booter chog doit
  ext2fs gen initrd-tree initrd.gz
==> found initrd.gz : Loading initrd : rdstart = 661 rdsize = 13523647 bytes
.....
.....
ready to go?
```

Figure 3.19. CDOM iso-booter Screen

Example 6. Linux LiveCD: We can boot up a Linux kernel from a CD and let it run on the CD directly. First, create a CD containing a base Linux system. Install the iso-booter on the CD to boot up a generic Linux kernel with an initrd image. While running on the ramdisk, load the isofs driver module. Then mount the CD and switch root file system to the CD. Linux would run on the booting CD directly (albeit in read-only mode). This is the basis of what's commonly known as a Linux LiveCD. For more information, the reader may consult the numerous LiveCD sites on the Internet.

Example 7. MTX Install CD: The iso-booter is the booter of MTXinstallCD.iso. It boots up a generic Linux kernel (version 2.6) and loads an initial RAM disk image, initrd.gz. When Linux boots up, it runs on the RAM disk image, which is used to install MTX from the CDROM.

3.6. USB Drive Booter

USB drives are storage devices connected to the USB bus. From a user point of view, USB drives are similar to hard disks. A USB drive can be divided into partitions. Each partition can be formatted as a unique file system and installed with a different operating system. To make a USB drive bootable, we install a booter to the USB drive's MBR and configure BIOS to boot from the USB drive. On some PCs, e.g. HP and Compaq, the USB drive must have a bootable partition marked as active. During booting, BIOS

emulates the booting USB drive as C: drive. The booting actions are exactly the same as that of booting from a hard disk. The booted up environment is also the same as if booted up from the first hard disk. Any booter that works for hard disk should also work for USB drives. Therefore, USB booting is identical to hard disk booting. However, depending on the booted up OS, there may be a difference. Usually, an OS that boots up from a hard disk can run directly on the hard disk. This may not be true if the OS is booted up from a USB drive. In the following, we use two specific examples to illustrate the difference.

3.6.1. MTX on USB Drive

MTX.bios is a MTX system which can be installed and run on either a floppy disk or a hard disk partition. In MTX.bios, all I/O operations are based on BIOS. When running on a FD, it uses BIOS INT13 to read-write floppy disk in CHS format. When running on a HD, it uses INT13-42 to read-write hard disk sectors in LBA. The following example shows how to install and run MTX.bios on a USB drive.

Example 7. /BOOTERS/USB/usbmtx demonstrates running MTX on a USB drive, denoted by /dev/sda. If the PC's HD is a SATA drive, change the USB drive to /dev/sdb. First, run the sh script, install.usb.sh, to install MTX to a USB drive partition.

```
mke2fs /dev/sda3 -b 1024 8192 # assume USB drive partition 3
mount /dev/sda3 /mnt
mount -o loop MTX.bios /tmp # mount MTX.bios
cp -av /tmp/* /mnt
umount /mnt; umount /tmp
```

Next, install hd-booter to the USB drive by

```
dd if=hd-booter of=/dev/sda bs=16 count=27
dd if=hd-booter of=/dev/sda bs=16 seek=1
```

Then boot from the USB drive under QEMU, as in

```
qemu -hda /dev/sda
```

When the MTX kernel starts, it can access the USB drive through BIOS INT13-42 on drive number 0x80. Since the environment is exactly the same as if running on a hard disk, MTX will run on the USB drive.

3.6.2. Linux on USB Drive

The last booting example is to create a so called "Linux Live USB". A live USB refers to a USB drive containing a complete operating system, which can boot up and run on the USB drive directly. Due to its portability and convenience, Linux live USB has received much attention and generated a great deal of interests among Linux users in recent years. The numerous "Linux live USB" sites and postings on the Internet attest to its popularity. When it first started in the late 90's the storage capacity of USB drives was relatively small. The major effort of earlier work was to create "small" Linux systems that can fit into USB drives. As the storage capacity of USB drives increases, this is no longer a restriction. At the time of this writing, 32 to 64 GB USB drives are very common and affordable. It is now possible to install a full featured Linux system on a USB drive still with plenty free space for applications and data. Most Linux live USB installations seem

to require a special setup environment, such as Linux running on a live CD. This example is intended to show that it is fairly easy to create a Linux live USB from a standard Linux distribution package. To be more specific, we shall again use Slackware Linux 13.1, which is based on Linux kernel version 2.6.33.4, as an example. The Slackware Linux distribution package consists of several CDs or a single DVD disc. The steps to create a Linux live USB are as follows.

(1). Install Slackware 13.1 to a USB drive partition. Slackware 13.1 uses /dev/sda as the primary hard disk. The USB drive should be named /dev/sdb. Install Linux to a USB partition, e.g. /dev/sdb1, by following the installation procedures. Since our hd-booter can boot from both EXT3 and EXT4 partitions, the reader may choose either EXT3 or EXT4 file system.

(2). After installing Linux, the reader may install LILO as the Linux booter. However, when the Linux kernel boots up, it will fail to run because it cannot mount the USB partition (8,17) as root device. As in CDROM booting, the missing link is again an initial RAM disk image. The Linux kernel must run on a ramdisk first in order for it to install the needed USB drivers and activate the USB drive. So the problem is how to create such an initrd.gz file.

(3). While still in the installation environment, the USB partition is mounted on /mnt, which already has all the Linux commands and driver modules. Enter the following commands or run the commands as a sh script.

```
cd /mnt;      chroot /mnt                      # change root to /mnt
# create initrd-tree with USB drivers
mkinitrd -c -k 2.6.33.4 -f ext4 -r /dev/sdb1 -m crc16: \
          jbd2:mbcache:ext4:usb-storage:ehci-hcd:uhci-hcd:ohci-hcd
echo 10 > /boot/initrd-tree/wait-for-root # write to wait-for-root
mkinitrd                                           # generate initrd.gz again
# if install lilo as booter
echo "initrd = /boot/initrd.gz" >> /etc/lilo.conf # append lilo.conf
lilo                                              # install lilo again
# install hd-booter to USB drive
# dd if=hd-booter of=/dev/sda bs=16 count=27
# dd if=hd-booter of=/dev/sda bs=512 seek=1
```

The above commands create a /boot/initrd.gz with USB driver modules in /boot of the USB partition. In the directory /boot/initrd-tree/ created by mkinitrd, the default value of wait-for-root is 1 second, which is too short for USB drives. If the value is too small, initrc may be unable to mount the USB drive, leaving the Linux kernel stuck on the initial ramdisk. Change it to a larger value, e.g. 10, to let the initrc process wait for 10 seconds before trying to mount the USB partition. After booting up Linux, the reader may try different delay values to suit the USB drive. Instead of LILO, the reader may also install the hd-booter to the USB drive. Mount a CDROM or another USB drive containing the above sh script and the hd-booter. Run the above sh script and install the hd-booter by un-commenting the last two lines. Then boot from the USB drive. Linux should come up and run on the USB drive.

3.7. Listing of Booter Programs

All the booters developed in this chapter have been tested on both real PCs and many versions of virtual machines. The booter programs are in the BOOTERS directory on the MTX install CD. Figure 3.20 shows a complete list of the booter programs.

FD—	—loadSector	:	linuxSector, linux.sector.ramdisk, OneFDlinux, linux.cylinder, mtxSector
	—loadBlock	:	linuxBlock, mtxBlock
	—FS	:	linuxFS, mtxFS
HD—	—MBR.ext4	:	hd-booter for EXT2/3/4 file systems
CD—	—emulation	:	emuFD, emuHD
	—no-emulation	:	isobooter; linuxCD, mtxCD, slackCD
USB—	—HOWTOusblinux,		usbmtx

Figure 3.20. List of Booter Programs

Problems

1. FD booting:

(1). Assume that a one-segment program is running in the segment 0x1000.

What must be the CPU's segment registers?

(2). When calling BIOS to load FD sectors into memory, how to specify the memory address?

(3). In `getblk(u16 blk, char buf[])`, the CHS parameters are computed as

$(C, H, S) = ((2 * blk) / 36, ((2 * blk) \% 36) / 18, ((2 * blk) \% 36) \% 18);$

The conversion formula can be simplified, e.g. $C = blk / 18$, etc. Try to simplify the CHS expression. Write a C program to verify that your simplified expressions are correct, i.e. they generates the same (C,H,S) values as the original algorithm.

2. Assume: The loading segment of MTX is 0x1000. During booting, BIOS loads the first 512 bytes of a 1KB MTX booter to the segment 0x07C0. The booter should run right where it is first loaded, i.e. in the segment 0x07C0 without relocation.

(1). What must the booter do first?

(2). How to set the CPU's segment registers?

(3). What's the maximum run-time image size of the booter?

3. On the MTX install CD, `OneFDlinux.img` is an EXT2 file system containing a bootable Linux zImage in the `/boot` directory.

(1). Using it as a virtual FD, verify that Linux can boot up and run on the same FD.

(2). Replace the booter in Block 0 with a suitable Linux booter developed in this chapter.

4. Prove that when loading FD sectors into memory, we can load at most 4 consecutive sectors without crossing either cylinder or 64KB boundaries.

5. Modify the FD booter that uses the cross-country algorithm to load by tracks.
6. Ramdisk Programming: Assume: A MTX boot FD contains
|booter|MTX kernel image|ramdiskImage|
where ramdiskImage in the last 128 blocks is a root file system for the MTX kernel. When the MTX kernel starts, it loads the ramdiskImage to the segment 0x8000. Then the MTX kernel uses the memory area between 0x8000 to 0xA000 as a ramdisk and runs on the ramdisk. Write C code for the functions
getblk(u16 blk, char buf[1024]) / putblk(u16 blk, char buf[1024])
which read/write a 1KB block from/to the ramdisk. HINT: use get_word()/put_word().
7. Under Linux, write a C program to print all the partitions of a hard disk.
8. Write a C program, showblock, which prints all the disk block numbers of a file in an EXT4 file system. The program should run as follows.
showblock DEVICE PATHNAME
e.g. showblock /dev/sda2 /a/b/c/d # /dev/sda2 for SATA hard disk partition 2
9. Assume that /boot/osimage is a bootable OS image. Write a C program, which finds the disk blocks of the OS image and store them in a /osimage.blocks file. Then write a booter, which simply loads the disk blocks in the /osimage.blocks file. Such a booter may be called an offline booter. The Linux boot-loader, LILO, uses this scheme. Discuss the advantages and disadvantages of off-line booters.
10. When booting a Linux bzImage, if the Linux kernel does not begin at a block boundary, loading the Linux kernel is rather complex. Given a Linux bzImage, devise a scheme which makes the Linux kernel always begin at a block boundary.
12. Modify the hd-booter to accept input parameters. For example, when the booter starts, the user may enter an input line
kernel=/boot/newvmlinuz initrd=/boot/initrd.gz root=/dev/sda7
where each parameter is of the form KEYWORD=value.
13. Modify the hd-booter to allow symbolic-link filenames for initrd.gz in the hd-booter.
14. The iso-booter does not handle symbolic-link filenames. Modify the C code to allow symbolic links.
15. Use the path table of an iso9660 file system to find the parent directory record.
16. USB booting: In some USB drives, a track may have less than 20 sectors. The hd-booter size is just over 10KB. How to install the hd-booter to such USB drives? Use the hd-booter and a Linux distribution package, e.g. Slackware 14.0, to create a Linux Live USB.

17. The ultimate version of MTX supports SMP in 32-bit protected mode, which is developed in Chapter 15 of this book. The bootable image of a SMP_MTX is a file consisting of the following pieces:

```

Sector  0      1      2      3  |  4      .....
-----
|BOOT|SETUP|  APentry  | MTX kernel                |
-----

```

where APentry is the startup code of non-boot processors in a SMP system. During booting, the booter loads BOOT+SETUP to 0x90000, APentry to 0x91000, and the MTX kernel to 0x10000. After loading completes, it sends the CPU to execute SETUP at 0x90200. Modify the MTX booter for booting SMP_MTX images.

18. Write a loader for loading a.out files into memory for execution. When loading an a.out file, it is more convenient to load the file by blocks. Assume that a one-segment a.out file (with header) is loaded at the segment address 0x2000, and it should run in that segment.

- (1). How to set up the CPU's segment registers?
- (2). Show how to eliminate the 32-byte header after loading a.out to a segment.

References

1. BusyBox: www.busybox.net
2. GNU GRUB Project: www.gnu.org/software/grub/
3. Cao, M., Bhattacharya, S, Tso, T., "Ext4: The Next Generation of Ext2/3 File system", IBM Linux Technology Center, 2007.
4. Comer, D.E., "Internetworking with TCP/IP: Principles, Protocols, and Architecture, 3/E", Prentice-Hall, 1995.
5. Comer,D.E., Stevens,D.L., "Internetworking With TCP/IP: Design, Implementation, and Internals, 3/E", Prentice-Hall, 1998.
6. Jones, M.T, "Linux initial RAM disk (initrd) overview", IBM developerworks, linux, Technical library, 2006
7. Slakware Linux: slackware.osuosl.org/slackware/README.initrd
8. Standard ECMA-119, Volume and File Structure of CDROM for Information Interchange,2nd edition, December 1987.
9. Syslinux project, www.syslinux.org
10. Stevens, C.E, Merkin, S. The "El Torito" Bootable CD-ROM Format Specification, Version 1.0, January, 1995