Chapter 3
Booting Operating Systems

3.1 Booting

Booting, which is short for bootstrap, refers to the process of loading an operating system image into computer memory and starting up the operating system. As such, it is the first step to run an operating system. Despite its importance and widespread interests among computer users, the subject of booting is rarely discussed in operating system books. Information on booting are usually scattered and, in most cases, incomplete. A systematic treatment of the booting process has been lacking. The purpose of this chapter is to try to fill this void. In this chapter, we shall discuss the booting principle and show how to write booter programs to boot up real operating systems. As one might expect, the booting process is highly machine dependent. To be more specific, we shall only consider the booting process of Intel x86 based PCs. Every PC has a BIOS (Basic Input Output System) program stored in ROM (Read Only Memory). After power on or following a reset, the PC's CPU starts to execute BIOS. First, BIOS performs POST (Power-on Self Test) to check the system hardware for proper operation. Then it searches for a device to boot. Bootable devices are maintained in a programmable CMOS memory. The usual booting order is floppy disk, CDROM, hard disk, etc. The booting order can be changed through BIOS. If BIOS finds a bootable device, it tries to boot from that device. Otherwise, it displays a "no bootable device found" message and waits for user intervention.

3.1.1 Bootable Devices

A bootable device is a storage device supported by BIOS for booting. Currently, bootable devices include floppy disk, hard disk, CD/DVD disc and USB drive. As storage technology evolves, new bootable devices will undoubtedly be added to the list, but the principle of booting should remain the same. A bootable device contains a booter and a bootable system image. During booting, BIOS loads the first 512 bytes of the booter to the memory location (segment, offset) = (0x0000, 0x7C00) = 0x07C00, and jumps to there to execute the booter. After that, it is

K. C. Wang, *Design and Implementation of the MTX Operating System,*
DOI 10.1007/978-3-319-17575-1_3

entirely up to the booter to do the rest. The reason why BIOS always loads the booter to 0x07C00 is historical. In the early days, a PC is only guaranteed to have 64 KB of RAM memory. The memory below 0x07C00 is reserved for interrupt vectors, BIOS and BASIC, etc. The first OS usable memory begins at 0x08000. So the booter is loaded to 0x07C00, which is 1 KB below 0x08000. When execution starts, the actions of a booter are typically as follows.

Load the rest of the booter into memory and execute the complete booter.
Find and load the operating system image into memory.
Send CPU to execute the startup code of the OS kernel, which starts up the OS.

Details of these steps will be explained later when we develop booter programs. In the following, we first describe the booting process of various bootable devices.

3.2 Booting from Various Devices

3.2.1 Floppy Disk Booting

As a storage device, floppy disk (FD) has almost become obsolete. Currently, most PCs, especially laptop computers, no longer support floppy drives. Despite this, it is still worth discussing FD booting for several reasons. First, booting requires writing a booter to the beginning part of a device, such as sector 0 of a hard disk, which is known as the Master Boot Record (MBR). However, writing to a hard disk is very risky. A careless mistake may render the hard disk non-bootable, or even worse, destroy the disk's partition table with disastrous consequences. In contrast, writing to a floppy disk involves almost no risk at all. It provides a simple and safe tool for learning the booting process and testing new booter programs. This is especially beneficial to beginners. Second, floppy drives are still supported in almost all PC emulators, such as QEMU, VMware and VirtualBox, etc. These PC emulators provide a virtual machine environment for developing and testing system software. Virtual machines are more convenient to use since booting a virtual machine does not need to turn off/on a real computer and the booting process is also faster. Third, for various reasons a computer may become non-bootable. Often the problem is not due to hardware failure but corrupted or missing system files. In these situations it is very useful to have an alternative way to boot up the machine to repair or rescue the system. Depending on the disk contents, FD booting can be classified into several cases, as shown in Fig. 3.1. In the following, we shall describe the setup and booting sequence of each case.

(1). FD contains a booter followed by a bootable OS image: In this case, a FD is dedicated to booting. It contains a booter in sector 0, followed by a bootable OS image in consecutive sectors, as shown in Fig. 3.2.

The size of the OS image, e.g. number of sectors, is either in the beginning part of the OS image or patched in the booter itself, so that the booter can determine how many sectors of the OS image to load. The loading address is also known, usually

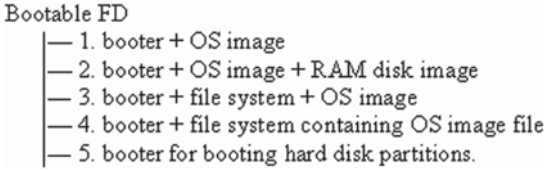


Fig. 3.1 Bootable FDs by Contents

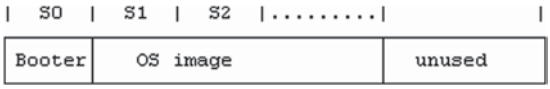


Fig. 3.2 A simple bootable FD layout

by default. In this case the booter's task is very simple. All it needs to do is to load the OS image sectors to the specified address and then send the CPU to execute the loaded OS image. Such a booter can be very small and fit easily in the MBR sector. Examples of this kind of setup include MTX, MINIX and bootable FD of small Linux zImage kernel.

(2). FD with a bootable image and a RAM disk Image: Although it is very easy to boot up an OS kernel from a FD, to make the OS kernel runnable is another matter. In order to run, most OS kernels require a root file system, which is a basic file system containing enough special files, commands and shared libraries, etc. that are needed by the OS kernel. Without a root file system, an OS kernel simply cannot run. There are many ways to make a root file system available. The simplest way is to assume that a root file system already exists on a separate device. During booting the OS kernel can be instructed to mount the appropriate device as the root file system. As an example, early distributions of Linux used a pair of boot-root floppy disks. The boot disk is used to boot up the Linux kernel, which is compiled with RAM disk support. The root disk is a compressed RAM disk image of a root file system. When the Linux kernel starts up, it prompts and waits for a root disk to be inserted. When the root disk is ready, the kernel loads the root disk contents to a ram disk area in memory, un-compresses the RAM disk image and mounts the ram disk as the root file system. The same boot-root disk pair was used later as a Linux rescue system.

If the OS and RAM disk images are small, it is possible to put both images on the same floppy disk, resulting in a single-FD system. Figure 3.3 shows the layout

Fig. 3.3 Bootable FD with RAM disk image

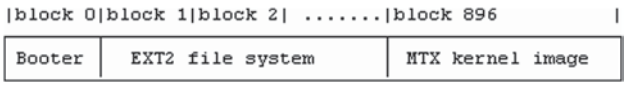


Fig. 3.4 FD with EXT2 file system and MTX kernel

of such a disk. It contains a booter and a Linux zImage followed by a compressed RAM disk image. The Linux kernel's ramdisk parameter can be set in such a way that when the Linux kernel starts, it does not prompt for a separate root disk but loads the RAM disk image directly from the boot disk.

Instead of a RAM disk image, a FD may contain a complete file system in front, followed by an OS image in the latter part of the disk. Figure 3.4 shows the layout of a single-FD real mode MTX system.

The real mode MTX image size is at most 128 KB. We can format a FD as an EXT2 file system with 1024 − 128 = 896 blocks, populate it with files needed by the MTX kernel and place the MTX kernel in the last 128 blocks of the disk. Block 0, which is not used by the file system, contains a MTX booter. During booting, the booter loads the MTX kernel from the last 128 disk blocks and transfers control to the MTX kernel. When the MTX kernel starts, it mounts the FD as the root file system. For small Linux zImage kernels, a single-FD Linux system is also possible.

(3). FD is a file system with bootable image files: In this case, the FD is a complete file system containing a bootable OS image as a regular file. To simplify booting, the OS image can be placed directly under the root directory, allowing the booter to find it easily. It may also be placed anywhere in the file system, e.g. in a /boot directory. In that case, the booter size would be larger since it must traverse the file system to find the OS image. During booting, the booter first finds the OS image file. Then it loads the image's disk blocks into memory and sends the CPU to execute the loaded OS image. When the OS kernel starts, it mounts the FD as the root file system and runs on the same FD. This kind of setup is very common. A well-known example is DOS, which can boot up and run from the same FD. Here, we describe two specific systems based on MTX and Linux.

A FD based MTX system is an EXT2 file system. It contains all the files needed by the MTX kernel. Bootable MTX kernels are files in the /boot directory. Block0 of the disk contains a MTX booter. During booting, the booter prompts for a MTX kernel to boot. The default is mtx but it can be any file name in the /boot directory. With a bootable file name, the booter finds the image file and loads its disk blocks to the segment 0x1000. When loading completes, it transfers control to the kernel image. When the MTX kernel starts up, it mounts the FD as the root file system and runs on the same FD. Similarly, we can create a single-FD Linux system as follows.

Format a FD as EXT2 file system (mke2fs on /dev/fd0).
Mount the FD and create directories (bin, boot,dev, etc,lib, sbin,usr).
Populate the file system with files needed by the Linux kernel.
Place a Linux zImage with rootdev = (2,0) in the /boot directory.
Install a linux booter to block0 of the FD.

After booting up, the Linux kernel can mount the FD as the root file system and run on the same FD. Although the principle is simple, the challenge is how to make a complete Linux file system small enough to fit in a single FD. This is why earlier Linux had to use a separate root disk. The situation changed when a small Linux file system, called the BusyBox (BusyBox), became available. A small BusyBox is only about 400 KB, yet it supports all the basic commands of Unix, including a sh and a text editor that emulates both vi and emacs, an incredible feat indeed. As an example, Fig. 3.13 shows the screen of booting up and running a single-FD Linux system. In addition to supporting small stand-alone Linux systems, BusyBox has become the core of almost all Linux distribution packages for installing the Linux system.

(4). FD with a booter for HD booting: This is a FD based booter for booting from hard disk partitions. During booting, the booter is loaded from a FD. Once execution starts, all the actions are for booting system images from hard disk partitions. Since the hard disk is accessed in read-only mode, this avoids any chances of corrupting the hard disk. In addition, it also provides an alternative way to boot up a PC when the normal booter becomes inoperative.

3.2.2 Hard Disk Booting

The discussion here is based on IDE hard disks but the same principle also applies to SCSI and SATA hard disks.

(1). Hard Disk Partitions: A hard disk is usually divided into several partitions. Each partition can be formatted as a unique file system and contain a different operating system. The partitions are defined by a partition table in the first (MBR) sector of the disk. In the MBR the partition table begins at the byte offset 0x1BE (446). It has four 16-byte entries for four primary partitions. If needed, one of the partitions can be EXTEND type. The disk space of an EXTEND partition can be further divided into more partitions. Each partition is assigned a unique number for identification. Details of the partition table will be shown later. For the time being, it suffices to say that from the partition table, we can find the start sector and size of each partition. Similar to the MBR, the first sector of each partition is also a (local) MBR, which may contain a booter for booting an OS image in that partition. Figure 3.5 shows the layout of a hard disk with partitions.

(2). Hard Disk Booting Sequence: When booting from a hard disk BIOS loads the MBR booter to the memory location (0x0000, 0x7C00) and executes it as usual.

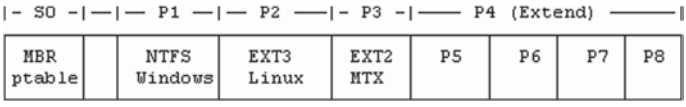


Fig. 3.5 Hard disk partitions

What happens next depends on the role of the MBR booter. In the simplest case, the MBR booter may ask for a partition to boot. Then it loads the local MBR of the partition to (0x0000, 0x7C00) and executes the local MBR booter. It is then up to the local MBR booter to finish the booting task. Such a MBR booter is commonly known as a chain-boot-loader. It might as well be called a pass-the-buck booter since all it does is to usher in the next booter and says "you do it". Such a chain boot-loader can be very small and fit entirely in the MBR. On the other hand, a more sophisticated HD booter should perform some of the booting tasks by itself. For example, the Linux boot loader LILO can be installed in the MBR for booting Linux as well as DOS and Windows. Similarly, GRUB (GNU GRUB Project) and the hd-booter developed in this book can also be installed in the MBR to boot up different operating systems. In general, a MBR booter cannot perform the entire booting task by itself due to its small size and limited capability. Instead, the MBR booter is only the beginning part of a multi-stage booter. In a multi-stage booter, BIOS loads stage1 and executes it first. Then stage1 loads and executes stage2, which loads and executes stage3, etc. Naturally, each succeeding stage can be much larger and more capable than the preceding stage. The number of stages is entirely up to the designer's choice. For example, GRUB version 1 and earlier had a stage1 booter in MBR, a stage1.5 and also a stage2 booter. In the latest GRUB_2, stage1.5 is eliminated, leaving only two stages. In the hd-booter of this book, the MBR booter is also part of the second stage booter. So strictly speaking it has only one stage.

3.2.3 CD/DVD_ROM Booting

Initially, CDROMs are used mainly for data storage, with proprietary booting methods provided by different computer vendors. CDROM booting standard was added in 1995. It is known as the El-Torito bootable CD specification (Stevens and Merkin 1995). Legend has it that the name was derived from a Mexican restaurant in California where the two engineers met and drafted the protocol.

(1). The El-Torito CDROM boot protocol: The El-Torito protocol supports three different ways to set up a CDROM for booting, as shown in Fig. 3.6.

(2). Emulation Booting: In emulation booting, the boot image must be either a floppy disk image or a (single-partition) hard disk image. During booting, BIOS loads the first 512 bytes of a booter from the boot image to (0x0000, 0x07C0) and execute the booter as usual. In addition, BIOS also emulates the CD/DVD drive as either a FD or HD. If the booting image size is 1.44 or 2.88 MB, it emulates the CD/

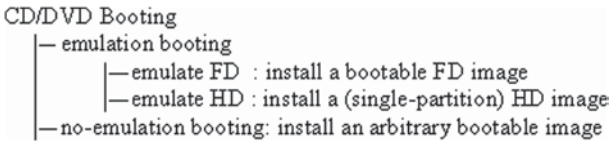


Fig. 3.6 CD/DVD boot options

DVD as the first floppy drive. Otherwise, it emulates the CD/DVD as the first hard drive. Once boot up, the boot image on the CD/DVD can be accessed as the emulated drive through BIOS. The environment is identical to that of booting up from the emulated drive. For example, if the emulated boot image is a FD, after booting up the bootable FD image can be accessed as A: drive, while the original A: drive is demoted to B: drive. Similarly, if the boot image is a hard disk image, after booting up the image becomes C: drive and the original C: drive becomes D: drives, etc. Although the boot image is accessible, it is important to note that nothing else on the CD/DVD disc is visible at this moment. This implies that, even if the CD/DVD contains a file system, the files are totally invisible after booting up, which may be somewhat surprising. Naturally, they will become accessible if the booted up kernel has a CD/DVD driver to read the CD/DVD contents.

(3). No-emulation Booting: In no-emulation booting, the boot image can be any (real-mode) binary executable code. For real-mode OS images, a separate booter is not necessary because the entire OS image can be booted into memory directly. For other OS images with complex loading requirements, e.g. Linux, a separate OS booter is needed. During booting, the booter itself can be loaded directly, but there is a problem. When the booter tries to load the OS image, it needs a device number of the CD/DVD drive to make BIOS calls. The question is: which device number? The reader may think it would be the usual device number of the CD/DVD drive, e.g. 0x81 for the first IDE slave or 0x82 for the second IDE master, etc. But it may be none of the above. The El-Torito protocol only states that BIOS shall emulate the CD/DVD drive by an arbitrary device number. Different BIOS may come up with different drive numbers. Fortunately, when BIOS invokes a booter, it also passes the emulated drive number in the CPU's DL register. The booter must catch the drive number and use it to make BIOS calls. Similar to emulation booting, while it is easy to boot up an OS image from CD/DVD, to access the contents on the CD/DVD is another matter. In order to access the contents, a booted up OS must have drivers to interpret the iso9660 file system on the CD/DVD.

3.2.4 USB Drive Booting

As a storage device, USB drives are similar to hard disks. Like a hard disk, a USB drive can be divided into partitions. In order to be bootable, some BIOS even require a USB drive to have an active partition. During booting, BIOS emulates the USB drive as the usual C: drive (0x80). The environment is the same as that of booting from the first hard disk. Therefore, USB booting is identical to hard disk booting.

As an example, it is very easy to install Linux to a USB partition, e.g. partition 2 of a USB drive, and then install LILO or GRUB to the USB's MBR for booting Linux from the USB partition. The procedure is exactly the same as that of installing Linux to a hard disk partition. If the PC's BIOS supports USB booting, Linux kernel will boot up from the USB partition. However, after boot up, the Linux kernel will fail to run because it can not mount the USB partition as root device, even

if all the USB drivers are compiled into the Linux kernel. This is because, when the Linux kernel starts, it only activates drivers for IDE and SCSI devices but not for USB drives. It is certainly possible to modify Linux's startup code to support USB drives, but doing so is still a per-device solution. Instead, Linux uses a general approach to deal with this problem by using an initial RAM disk image.

3.2.5 Boot Linux with Initial Ramdisk Image

Booting Linux kernel with an initial RAM disk image has become a standard way to boot up a Linux system. The advantage of using an initrd image is that it allows a single generic Linux kernel to be used on many different Linux configurations. An initrd is a RAM disk image which serves as a temporary root file system when the Linux kernel first starts up. While running on the RAM disk, the Linux kernel executes a sh script, initrc, which directs the kernel to load the driver modules of the real root device, such as a USB drive. When the real root device is activated and ready, the kernel discards the RAM disk and mounts the real root device as the root file system. Details of how to set up and load initrd image will be shown later.

3.2.6 Network Booting

Network booting has been in use for a long time. The basic requirement is to establish a network connection to a server machine in a network, such as a server running the BOOTP protocol in a TCP/IP (Comer 1995; Comer and Stevens 1998) network. Once the connection is made, booting code or the entire kernel code can be downloaded from the server to the local machine. After that, the booting sequence is basically the same as before. Networking is outside the scope of this book. Therefore, we shall not discuss network booting.

3.3 Develop Booter Programs

In this section, we shall show how to write booter programs to boot up real operating systems. Although this book is primarily about MTX, we shall also discuss Linux booting, for a number of reasons. First, Linux is a popular OS, which has a very large user base, especially among Computer Science students. A Linux distribution usually comes with a default booter, which is either LILO or GRUB. After installing Linux, it would boot up nicely. To most users the booting process remains somewhat a mystery. Many students often wish to know more about the booting process. Second, Linux is a very powerful real OS, which runs on PCs with a wide range of hardware configurations. As a result, the booting process of Linux is also fairly complex and demanding. Our purpose is to show that, if we can write a booter

to boot up Linux, we should be able to write a booter to boot up any operating system. In order to show that the booters actually work, we shall demonstrate them by sample systems. All the booter programs developed in this chapter are in the MTX. src/BOOTERS directory on the MTX install CD. A detailed listing of the booter programs is at the end of this chapter.

3.3.1 Requirements of Booter Programs

Before developing booter programs, we first point out the unique requirements of booter programs.

1. A booter needs assembly code because it must manipulate CPU registers and make BIOS calls. Many booters are written entirely in assembly code, which makes them hard to understand. In contrast, we shall use assembly code only if absolutely necessary. Otherwise, we shall implement all the actual work in the high-level language C.
2. When a PC starts, it is in the 16-bit real mode, in which the CPU can only execute 16-bit code and access the lowest 1 MB memory. To create a booter, we must use a compiler-linker that generates 16-bit code. For example, we can not use GCC because the GCC compiler generates 32 or 64-bit code, which can not be used during booting. The software chosen is the BCC package under Linux, which generates 16-bit code.
3. By default, the binary executable generated by BCC uses a single-segment memory model, in which the code, data and stack segments are all the same. Such a program can be loaded to, and executed from, any available segment in memory. A segment is a memory area that begins at a 16-byte boundary. During execution, the CPU's CS, DS and SS registers must all point to the same segment of the program.
4. Booters differs from ordinary programs in many aspects. Perhaps the most notable difference is their size. A booter's size (code plus static data) is extremely limited, e.g. 512 or 1024 bytes, in order to fit in one or two disk sectors. Multistage booters can be larger but it is always desirable to keep the booter size small. The second difference is that, when running an ordinary program an operating system will load the entire program into memory and set up the program's execution environment before execution starts. An ordinary program does not have to worry about these things. In contrast, when a booter starts, it only has the first 512 bytes loaded at 0x07C00. If the booter is larger than 512 bytes, which is usually the case, it must load the missing parts in by itself. If the booter's initial memory area is needed by the OS, it must be moved to a different location in order not to be clobbered by the incoming OS image. In addition, a booter must manage its own execution environment, e.g. set up CPU segment registers and establish a stack.
5. A booter cannot use the standard library I/O functions, such as gets() and printf(), etc. These functions depend on operating system support, but there is no operat-

ing system yet during booting. The only available support is BIOS. If needed, a booter must implement its own I/O functions by calling only BIOS.

6. When developing an ordinary program we may use a variety of tools, such as gdb, for debugging. In contrast, there is almost no tool to debug a booter. If something goes wrong, the machine simply stops with little or no clue as to where and why the error occurred. This makes writing booter programs somewhat harder. Despite these, it is not difficult to write booter programs if we follow good programming practice.

3.3.2 Online and Offline Booters

There are two kinds of booters; online and offline. In an offline booter, the booter is told which OS image (file) to boot. While running under an operating system, an offline booter first finds the OS image and builds a small database for the booter to use. The simplest database may contain the disk blocks or ranges of disk blocks of the OS image. During booting, an offline booter simply uses the pre-built database to load the OS image. For example, the Linux boot-loader, LILO, is an offline booter. It uses a lilo.conf file to build a map file in the /boot directory, and then installs the LILO booter to the MBR or the local MBR of a hard disk partition. During booting, it uses the map file in the /boot directory to load the Linux image. The disadvantage of offline booters is that the user must install the booter again whenever the OS image is moved or changed. In contrast, an online booter, e.g. GRUB, can find and load an OS image file directly. Since online booters are more general and flexible, we shall only consider online booters.

3.3.3 Boot MTX from FD sectors

We begin with a simple booter for booting MTX from a FD disk. The FD disk layout is shown in Fig. 3.7.

It contains a booter in Sector 0, followed by a MTX kernel image in consecutive sectors. In the MTX kernel image, which uses the separate I&D memory model,

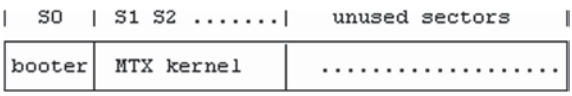


Fig. 3.7 MTX boot disk layout

the first three (2-byte) words are reserved. Word 0 is a jump instruction, word 1 is the code section size in 16-byte clicks and word 2 is the data section size in bytes. During booting, the booter may extract these values to determine the number of sectors of the MTX kernel to load. The loading segment address is 0x1000. The booter consists of two files, a bs.s file in BCC assembly and a bc.c file in C. Under Linux, use BCC to generate a binary executable without header and dump it to the beginning of a floppy disk, as in

```
as86 -o bs.o  bs.s      # assemble bs.s into bs.o
bcc  -c -ansi bc.c      # compile  bc.c into bc.o
# link bs.o and bc.o into a binary executable without header
ld86 -d -o booter bs.o bc.o /usr/lib/bcc/libc.a
# dump booter to sector 0 of a FD
dd if=booter of=/dev/fd0 bs=512 count=1 conv=notrunc
```

where the special file name, /dev/fd0, is the first floppy drive. If the target is not a real device but an image file, simply replace /dev/fd0 with the image file name. In that case, the parameter conv = notrunc is necessary in order to prevent dd from truncating the image file. Instead of entering individual commands, the building process can be automated by using a Makefile or a sh script. For simple compile-link tasks, a sh script is adequate and actually more convenient. For example, we may re-write the above commands as a sh script file, mk, which takes a filename as parameter.

```
# usage: mk filename
as86 -o bs.o bs.s    # bs.s file does not change
bcc  -c -ansi $1.c
ld86 -d -o $1 bs.o $1.o /usr/lib/bcc/libc.a
dd if=$1 of=IMAGE bs=512 count=1 conv=notrunc
```

In the following, we shall assume and use such a sh script. First, we show the booter's assembly code.

```
!======================== bs.s file ============================
.globl  _main,_prints,_NSEC                     ! IMPORT from C
.globl  _getc,_putc,_readfd,_setes,_inces, _error ! EXPORT to C
        BOOTSEG  =  0x9800  ! booter segment
        OSSEG    =  0x1000  ! MTX kernel segment
        SSP      =  32*1024 ! booter stack size=32KB
        BSECTORS =  2       ! number of sectors to load initially
! Boot SECTOR loaded at (0000:7C00). reload booter to segment 0x9800
start:
        mov  ax, #BOOTSEG   ! set ES to 0x9800
        mov  es, ax
! call BIOS INT13 to load BSECTORS to (segment,offset)=(0x9800,0)
        xor  dx, dx         ! dh=head=0, dl=drive=0
        xor  cx, cx         ! ch=cyl=0,  cl=sector=0
        incb cl             ! sector=1 (BIOS counts sector from 1)
        xor  bx, bx         ! (ES,BX)= real address = (0x9800,0)
        movb ah, #2         ! ah=READ
        movb al, #BSECTORS  ! al=number of sectors to load
        int  0x13           ! call BIOS disk I/O function
! far jump to (0x9800, next) to continue execution there
        jmpi next, BOOTSEG  ! CS=BOOTSEG, IP=next
next:
        mov  ax, cs         ! Set CPU segment registers to 0x9800
        mov  ds, ax         ! we know ES=CS=0x9800. Let DS=CS
        mov  ss, ax         ! let SS = CS
        mov  sp, #SSP       ! SP = SS + 32 KB
        call _main          ! call main() in C
        jmpi 0, OSSEG       ! jump to execute OS kernel at (OSSEG,0)
!==================== I/O functions ==========================
_getc: ! char getc(): return an input char
        xorb ah, ah         ! clear ah
        int  0x16           ! call BIOS to get a char in AX
        ret
_putc: ! putc(char c): print a char
        push bp
        mov  bp, sp
        movb al, 4[bp]      ! aL = char
        movb ah, #14        ! aH = 14
        int  0x10           ! call BIOS to display the char
        pop  bp
        ret
_readfd: ! readfd(cyl,head,sector): load _NSEC sectors to (ES,0)
        push bp
        mov  bp, sp         ! bp = stack frame pointer
        movb dl, #0x00      ! drive=0 = FD0
        movb dh, 6[bp]      ! head
        movb cl, 8[bp]      ! sector
        incb cl             ! inc sector by 1 to suit BIOS
        movb ch, 4[bp]      ! cyl
        xor  bx, bx         ! BX=0
        movb ah, #0x02      ! READ
        movb al, _NSEC      ! read _NSEC sectors to (ES,BX)
        int  0x13           ! call BIOS to read disk sectors
        jb   _error         ! error if CarryBit is set
        pop  bp
        ret
_setes:                     ! setes(segment): set ES to a segment
        push bp
        mov  bp, sp
        mov  ax, 4[bp]
        mov  es, ax
        pop  bp
        ret
_inces: ! inces(): increment ES by _NSEC sectors (in 16-byte clicks)
        mov  bx, _NSEC      ! get _NSEC in BX
        shl  bx, #5         ! multiply by 2**5 = 32
        mov  ax, es         ! current ES
        add  ax, bx         ! add (_NSEC*0x20)
        mov  es, ax         ! update ES
        ret
_error:                     ! error() and reboot
        push #msg
        call _prints
        int  0x19           ! reboot
msg:    .asciz  "Error"
```

In the assembly code, start: is the entry point of the booter program. During booting, BIOS loads sector 0 of the boot disk to (0x0000, 0x7C00) and jumps to there to execute the booter. We assume that the booter must be relocated to a different memory area. Instead of moving the booter, the code calls BIOS INT13 to load the first 2 sectors of the boot disk to the segment 0x9800. The FD drive hardware can load a complete track of 18 sectors at a time. The reason of loading 2 (or more) sectors will become clear shortly. After loading the booter to the new segment, it does a far jump, jmpi next, 0x9800, which sets CPU's (CS, IP)=(0x9800, next), causing the CPU to continue execution from the offset next in the segment 0x9800. The choice of 0x9800 is based on a simple principle: the booter should be relocated to a high memory area with enough space to run, leaving as much space as possible in the low memory area for loading the OS image. The segment 0x9800 is 32 KB below the ROM area, which begins at the segment 0xA000. This gives the booter a 32 KB address space, which is big enough for a fairly powerful booter. When execution continues, both ES and CS already point to 0x9800. The assembly code sets DS and SS to 0x9800 also in order to conform to the one-segment memory model of the program. Then it sets the stack pointer to 32 KB above SS. Figure 3.8 shows the run-time memory image of the booter.

It is noted that, in some PCs, the RAM area above 0x9F000 may be reserved by BIOS for special usage. On these machines the stack pointer can be set to a lower address, e.g. 16 KB from SS, as long as the booter still has enough bss and stack space to run. With a stack, the program can start to make calls. It calls main() in C, which implements the actual work of the booter. When main() returns, it sends the CPU to execute the loaded MTX image at (0x1000, 0).

The remaining assembly code contains functions for I/O and loading disk sectors. The functions getc() and putc(c) are simple; getc() returns an input char from the keyboard and putc(c) displays a char to the screen. The functions readfd(), setes() and inces() deserve more explanations. In order to load an OS image, a booter must be able to load disk sectors into memory. BIOS supports disk I/O functions via INT13, which takes parameters in CPU registers:

```
DH=head(0-1),  DL=drive(0 for FD drive 0),
CH=cyl (0-79), CL=sector (1-18)
AH=2(READ),    AL=number of sectors to read
Memory address: (segment, offset)=(ES, BX)
return status : carry bit=0 means no error, 1 means error.
```

Fig. 3.8 Run-time image of booter

The function readfd(cyl, head, sector) calls BIOS INT13 to load NSEC sectors into memory, where NSEC is a global imported from C code. The zero-counted parameters, (cyl, head, sector), are computed in C code. Since BIOS counts sectors from 1, the sector value is incremented by 1 to suit BIOS. When loading disk sectors BIOS uses (ES, BX) as the real memory address. Since BX = 0, the loading address is (ES,0). Thus, ES must be set, by the setes(segment) function, to a desired loading segment before calling readfd(). The function code loads the parameters into CPU registers and issues INT 0x13. After loading NSEC sectors, it uses inces() to increment ES by NSEC sectors (in 16-byte clicks) to load the next NSEC sectors, etc. The error() function is used to trap any error during booting. It prints an error message, followed by reboot. The use of NSEC as an global rather than as a parameter to readfd() serves two purposes. First, it illustrates the cross reference of globals between assembly and C code. Second, if a value does not change often, it should not be passed as a parameter because doing so would increase the code size. Since the booter size is limited to 512 bytes, saving even a few bytes could make a difference between success and failure. Next, we show the booter's C code.

```
/****************** MTX booter's bc.c file **********************
 FD contains this booter in Sector 0, MTX kernel begins in Sector 1
 In the MTX kernel: word#1=tsize in clicks, word#2=dsize in bytes
*****************************************************************/
int tsize, dsize, ksectors, i, NSEC = 1;

int prints(char *s){  while(*s) putc(*s++); }

int getsector(u16 sector)
{  readfd(sector/36, ((sector)%36)/18, (((sector)%36)%18)); }

main()
{
  prints("booting MTX\n\r");
  tsize = *(int *)(512+2);
  dsize = *(int *)(512+4);
  ksectors = ((tsize << 4) + dsize + 511)/512;
  setes(0x1000);
  for (i=1; i<=ksectors+1; i++){
      getsector(i); inces(); putc('.');
  }
  prints("\n\rready to go?"); getc();
}
```

Explanations of C Code Disk sectors are numbered linearly as 0,1,2,…, but BIOS INT13 only accepts disk parameters in (cyl, head, sector) or CHS format. When calling BIOS INT13 we must convert the starting sector number into CHS format. Figure 3.9 shows the relationship between linear and CHS addressing of FD sectors.

Using the Mailman's algorithm, we can convert a linear sector number into CHS format as cyl = sec/36; head = (sec%36)/18; sector = (sec%36)%18;

Then write a getsector() function in C, which calls readfd() for loading disk sectors.

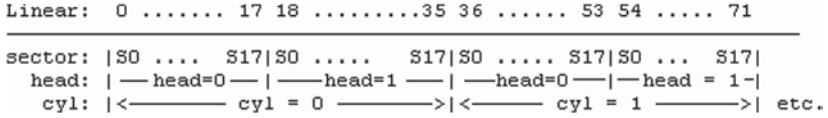


Fig. 3.9 Linear sector and CHS addressing

Fig. 3.10 Booting screen of MTX sector Booter

int getsector(int sec){ readfd(sec/36, (sec%36)/18, (sec%36)%18) }

In the C code, the prints() function is used to print message strings. It is based on putc() in assembly. As specified, on the boot disk the MTX kernel image begins from sector 1, in which word 1 is the tsize of the MTX kernel (in 16-byte clicks) and word 2 is the dsize in bytes. Before the booter enters main(), sectors 0 and 1 are already loaded at 0x9800. While in main(), the program's data segment is 0x9800. Thus, words 1 and 2 of sector 1 are now at the (offset) addresses 512+2 and 512+4, respectively. The C code extracts these values to compute the number of sectors of the MTX kernel to load. It sets ES to the segment 0x1000 and loads the MTX sectors in a loop. The loading scheme resembles that of a "sliding window". Each iteration calls getsector(i) to load NSEC sectors from sector i to the memory segment pointed by ES. After loading NSEC sectors to the current segment, it increments ES by NSEC sectors to load the next NSEC sectors, etc. Since NSEC = 1, this amounts to loading the OS image by individual sectors. Faster loading schemes will be discussed later is Sect. 3.3.5. Figure 3.10 shows the booting screen of the MTX.sector booter, in which each dot represents loading a disk sector.

3.3.4 Boot Linux zImage from FD Sectors

Bootable Limux images are generated as follows. Under Linux,

cd to linux source code tree directory (cd /usr/src/linux)
create a.config file, which guides make (make .config)
run make zImage to generate a small Linux image named zImage

Make zImage generates a small bootable Linux image, in which the compressed kernel size is less than 512 KB. In order to generate a small Linux zImage, we must select a minimal set of options and compile most of the device drivers as modules. Otherwise, the kernel image size may exceed 512 KB, which is too big to be loaded into real-mode memory between 0x10000 and 0x90000. In that case, we must use

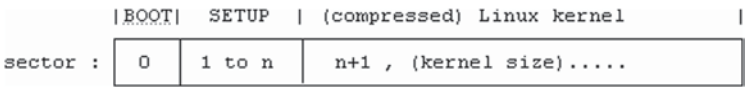


Fig. 3.11 Bootable linux image

make bzImage to generate a big Linux image, which requires a different loading scheme during booting. We shall discuss how to boot big Linux bzImages later. Regardless of size, a bootable Linux image is composed of three contiguous parts, as shown in Fig. 3.11.

where BOOT is a booter for booting Linux from floppy disk and SETUP is for setting up the startup environment of the Linux kernel. For small zImages, the number of SETUP sectors, *n*, varies from 4 to 10. In addition, the BOOT sector also contains the following boot parameters.

```
--------   ---------------------------------------
byte 497    number of SETUP sectors
byte 498    root dev flags: nonzero=READONLY
word 500    Linux kernel size in (16-byte) clicks
word 504    ram disk information
word 506    video mode
word 508    root device=(major, minor) numbers
-------------------------------------------------
```

Most of the boot parameters can be changed by the rdev utility program. The reader may consult Linux man page of rdev for more information. A zImage is intended to be a bootable FD disk of Linux. Since kernel version 2.6, Linux no longer supports FD booting. The discussion here applies only to small Linux zImages of kernel version 2.4 or earlier. During booting, BIOS loads the boot sector, BOOT, into memory and executes it. BOOT first relocates itself to the segment 0x9000 and jumps to there to continue execution. Then it loads SETUP to the segment 0x9020, which is 512 bytes above BOOT. Then it loads the Linux kernel to the segment 0x1000. When loading completes, it jumps to 0x90200 to run SETUP, which starts up the Linux kernel. The loading requirements of a Linux zImage are:

```
BOOT+SETUP   : 0x90000
Linux Kernel : 0x10000
```

Our Linux zImage booter essentially duplicates exactly what the BOOT sector does. A Linux zImage boot disk can be created as follows. First, use dd to dump a Linux zImage to a FD beginning in sector 1, as in

```
dd  if = zImage  of = /dev/fd0  bs = 512  seek =1  conv = notrunc
```

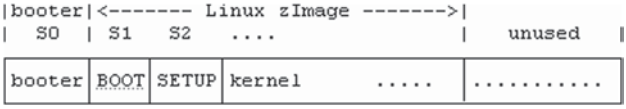


Fig. 3.12 Linux boot disk layout

Then install a Linux booter to sector 0. The resulting disk layout is shown in Fig. 3.12.

The MTX booter can be adapted to booting Linux from a zImage boot disk. In the assembly code, we only need to change OSSEG to 0x9020. When main() returns, it jumps to (0x9020, 0) to execute SETUP. The C code is almost the same as that of the MTX booter. We only show the modified main() function.

```
/***********  C code for Linux zImage booter **************/
int setup, ksectors, i;
main()
{
  prints("boot linux\n\r");
  setup = *(char *)(512+497);           // number of SETUP sectors
  ksectors = *(int *)(512+500) >> 5;    // number of kernel sectors
  setes(0x9000);                        // load BOOT+SETUP to 0x9000
  for (i=1; i<=setup+ksectors+2; i++){  // 2 sectors before SETUP
    getsector(i);                       // load sector i
    i <= setup ? putc('*') : putc('.'); // show a * or .
    inces();                            // inc ES by NSEC sector
    if (i==setup+1)                     // load kernel to ES=0x1000
       setes(0x1000);
  }
  prints("\n\ready to go?"); getc();
}
```

The booting screen of the Linux zImage booter is similar to Fig. 3.10, only with many more dots. In the Linux kernel image, the root device (a word at byte offset 508) is set to 0x0200, which is for the first FD drive. When Linux boots up, it will try to mount (2,0) as the root file system. Since the boot FD is not a file system, the mount will fail and the Linux kernel will display an error message, Kernel panic: VFS: Unable to mount root fs 02:00, and stop. To make the Linux kernel runnable, we may change the root device setting to a device containing a Linux file system. For example, assume that we have Linux installed in partition 2 of a hard disk. If we change the root device of zImage to (3,2), Linux would boot up and run successfully. Another way to provide a root file system is to use a RAM disk image. As an example, in the BOOTERS directory in the MTX install CD, OneFDlinux.img is a single-FD Linux image. It contains a Linux booter and a Linux zImage in front, followed by a compressed ramdisk image beginning in block 550. The Linux kernel is compiled with ramdisk support. The ramdisk

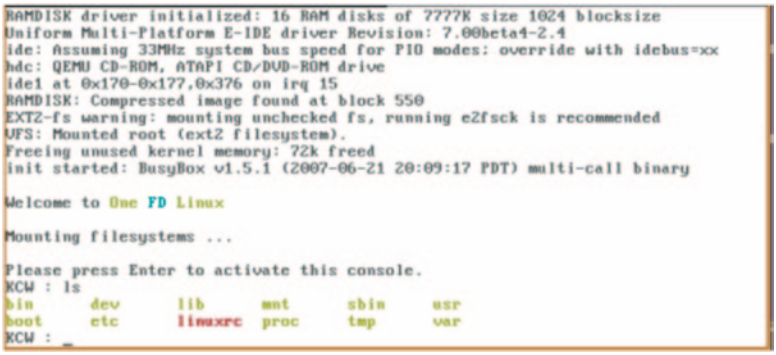


Fig. 3.13 Single-FD linux running on ramdisk

parameter is set to 16384 + 550 (bit14 = 1 plus ramdisk begin block), which tells the Linux kernel not to prompt for a separate ramdisk but load it from block 550 of the boot disk. Figure 3.13 shows the screen of running the single-FD Linux on a ramdisk.

3.3.5 Fast FD Loading Schemes

The above FD booters load OS images one sector at a time. For small OS images, such as the MTX kernel, this works fine. For large OS images like Linux, it would be too slow to be acceptable. A faster loading scheme is more desirable. When boot a Linux zImage, logically and ideally only two loading operations are needed, as in

```
setes(0x9000);  nsec = setup +1;  getsector(1);
setes(0x1000);  nsec = ksectors;  getsector(setup + 2);
```

Unfortunately, things are not so simple due to hardware limitations. The first problem is that FD drives cannot read across track or cylinder. All floppy drives support reading a full track of 18 sectors at a time. Some BIOS allows reading a complete FD cylinder of 2 tracks. The discussion here assumes 1.44 MB FD drives that support reading cylinders. When loading from FD the sectors must not cross any cylinder boundary. For example, from the sector number 34 (count from 0), read 1 or 2 sectors is OK but attempting to read more than 2 sectors would result in an error. This is because sectors 34 and 35 are in cylinder 0 but sector 36 is in cylinder 1; going from sector 35 to 36 crosses a cylinder boundary, which is not

allowed by the drive hardware. This means that each read operation can load at most a full cylinder of 36 sectors. Then, there is the infamous cross 64 KB boundary problem, which says that when loading FD sectors the real memory address cannot cross any 64 KB boundary. For example, from the real address 0x0FE00, if we try to load 2 sectors, the second sector would be loaded to the real address 0xFE000 + 0x200 = 0x10000, which crosses the 64 KB boundary at 0x10000. The cause of the problem is due to the DMA controller, which uses 18-bit address. When the low 16 bits of an address reaches 64K, for some reason the DMA controller does not increment the high order 2 bits of the address, causing the low 16-bit address to wrap around. In this case, loading may still occur but only to the same segment again. In the above example, instead of the intended address 0x10000, the second sector would be loaded to 0x00000. This would destroy the interrupt vectors, which effectively kills BIOS. Therefore, a FD booter must avoid both problems when loading an OS image. A simple way to avoid these problems is to load sectors one by one as we have done so far. Clearly, loading one sector at a time will never cross any cylinder. If the loading segment starts from a sector boundary, i.e. a segment address divisible by 0x20, it also will not cross any 64 KB boundary. Similarly, if the OS image starts from a block boundary on disk and the loading segment also starts from a block boundary in memory, then loading 1 KB blocks would also work. In order not to cross both cylinder and 64 KB boundaries, the best we can do is loading 4 sectors at a time. The reader is encouraged to prove this. Can we do better? The answer is yes, as evidenced by many published boot-loaders, most of which try to load by tracks. Here we present a fast loading scheme, called the “cross-country” algorithm, which loads by cylinders. The algorithm resembles a cross country runner negotiating an obstacle course. When there is open space, the runner takes full strides (load cylinders) to run fast. When there is an obstacle ahead, the runner slows down by taking smaller strides (load partial cylinder) until the obstacle is cleared. Then the runner resumes fast running by taking full strides, etc. The following C code shows a Linux zImage booter that implements the cross country algorithm. In order to keep the booter size within 512 bytes, updating ES is done inside getsector() and the prints() function is also eliminated. The resulting booter size is only 484 bytes.

```
/***************** Cross Country Algorithm *************************
Load cylinders. If a cylinder is about to cross 64KB, compute NSEC =
max sectors without crossing 64KB. Load NSEC sectors, load remaining
CYL-NSEC sectors. Then load cylinders again, etc.
*******************************************************************/
#define TRK 18
#define CYL 36
int setup, ksectors, ES;
int csector = 1;  // current loading sector
int NSEC = 35;  // initial number of sectors to load >= BOOT+SETUP
int getsector(u16 sector)
{
    readfd( sector/CYL,((sector)%CYL)/TRK,(((sector)%CYL)%TRK));
    csector += NSEC; inces();
}
main()
{
  setes(0x9000);
  getsector(1);             // load Linux's [boot+SETUP] to 0x9000
  // current sector = SETUP's sector count (at offset 512+497) + 2
  setup    = *(u8 *)(512+497) + 2;
  ksectors = (*(u16 *)(512+500)) >> 5;
  NSEC = CYL - setup;       // sectors remain in cylinder 0
  setes(0x1000);            // Linux kernel is loaded to segment 0x1000
  getsector(setup);         // load the remaining sectors of cylinder 0
  csector = CYL;            // we are now at begining of cyl#1
  while (csector < ksectors+setup){ // try to load cylinders
     ES = getes();          // current ES value
     if (((ES + CYL*0x20) & 0xF000) == (ES & 0xF000)){//same segment
        NSEC = CYL;         // load a full cylinder
        getsector(csector); putc('C'); // show loaded a cylinder
        continue;
     }
     // this cylinder will cross 64KB, compute MAX sectors to load
     NSEC = 1;
     while( ((ES + NSEC*0x20) & 0xF000) == (ES & 0xF000) ){
        NSEC++; putc('s'); // number of sectors can still load
     }                     // without crossing 64KB boundary
     getsector(csector);   // load partial cylinder
     NSEC = CYL - NSEC;    // load remaining sectors of cylinder
     putc('|');            // show cross 64KB
     getsector(csector);   // load remainder of cylinder
     putc('p');
  }
}
```

Figure 3.14 shows the booting screen of the linux.cylinder booter. In the figure, each C is loading a cylinder, each sequence of s is loading the sectors of a partial

Fig. 3.14 Booting linux by loading cylinders

cylinder, each | is crossing a 64 KB boundary and each p is loading the remaining sectors of a cylinder.

Instead of loading cylinders, the reader may modify the above program to load tracks. Similarly, the reader may modify the above booters to load disk blocks.

3.3.6 Boot MTX Image from File System

Our second booter is to boot MTX from a file system. A MTX system disk is an EXT2 file system containing files needed by the MTX kernel. Bootable MTX kernel images are files in the /boot directory. Block 0 of the disk contains the booter. The loading segment address is 0x1000. After booting up, the MTX kernel mounts the same boot disk as the root file system.

When booting an OS image from an EXT2 file system, the problem is essentially how to find the image file's inode. The reader may consult Sect. 2.8.2 of Chap. 2 for the algorithm. Here we only briefly review the steps. Assume that the file name is /boot/mtx. First, read in the 0th group descriptor to find the start block of the inodes table. Then read in the root inode, which is number 2 inode in the inode table. From the root inode's data blocks, search for the first component of the file name, boot. Once the entry boot is found, we know its inode number. Use Mailman's algorithm to convert the inode number to the disk block containing the inode and its offset in that block. Read in the inode of boot and repeat the search for the component mtx. If the search steps succeed, we should have the image file's inode in memory. It contains the size and disk blocks of the image file. Then we can load the image by loading its disk blocks.

When such a booter starts, it must be able to access the file system on the boot disk, which means loading disk blocks into the booter program's memory area. In order to do this, we add a parameter, buf, to the assembly function, readfd(char *buf), where buf is the address of a 1 KB memory area in the booter segment. It is passed to BIOS in BX as the offset of the loading address in the ES segment. Corresponding to this, we also modify getsector() in C to take a block number and buf as parameters. When the booter starts, ES points to the segment of the booter. In the booter's C code, if buf is global, it is relative to DS. If buf is local, it is relative to SS. Therefore, no matter how we define buf, the loading address is always in the booter segment. When loading the blocks of an OS image we can set ES to successive segments and use (ES, 0) as the loading address. The booter's assembly code is almost the same as before. We only show the booter's C code. The booter size is 1008 bytes, which can fit in the 1 KB boot block of a FD.

```
/*************************************************
*       Image file booter's bc.c code            *
*************************************************/
#include "ext2.h"  // contain EXT2 structure types
#define BLK 1024
typedef unsigned char  u8;
typedef unsigned short u16;
typedef unsigned long  u32;
typedef struct ext2_group_desc  GD;
typedef struct ext2_inode       INODE;
typedef struct ext2_dir_entry_2 DIR;
u16 NSEC = 2;
char buf1[BLK], buf2[BLK];     // 2 I/O buffers of 1KB each
int prints(char *s){ //same as before }
int gets(char *s){   // to keep code simple, no length checking
    while ((*s=getc()) != '\r')
          putc(*s++);
    *s = 0;
}
int getblk(u16 blk, char *buf)
{   readfd(blk/18, ((2*blk)%36)/18, ((2*blk)%36)%18, buf); }

u16 search(INODE *ip, char *name)
{
  int i;   char c;  DIR *dp;
  for (i=0; i<12; i++){ // assume a DIR has at most 12 direct blocks
      if ( (u16)ip->i_block[i] ){
         getblk((u16)ip->i_block[i], buf2);
         dp = (DIR *)buf2;
         while ((char *)dp < &buf2[BLK]){
            c = dp->name[dp->name_len];  // save last byte
            dp->name[dp->name_len] = 0;  // make name into a string
            prints(dp->name); putc(' '); // show dp->name string
            if ( strcmp(dp->name, name) == 0 ){
                prints("\n\r");
                return((u16)dp->inode);
            }
            dp->name[dp->name_len] = c;  // restore last byte
            dp = (char *)dp + dp->rec_len;
                                                                                    }
       }
  }
  error();  // to error() if can't find file name
}

main()  // booter's main function, called from assembly code
{
  char  *cp, *name[2], filename[64];
  u16   i, ino, blk, iblk;
  u32   *up;
  GD    *gp;
  INODE *ip;
  DIR   *dp;
  name[0] = "boot"; name[1] = filename;
  prints("bootname: ");
  gets(filename);
  if (filename[0]==0) name[1]="mtx";
  getblk(2, buf1);    // read blk#2 to get group descriptor 0
  gp = (GD *)buf1;
  iblk = (u16)gp->bg_inode_table;   // inodes begin block
  getblk(iblk, buf1);               // read first inode block
  ip = (INODE *)buf1 + 1;           // ip->root inode #2
  for (i=0; i<2; i++){              // serach for system name
      ino = search(ip, name[i]) - 1;
      if (ino < 0) error();         // if search() returned 0
      getblk(iblk+(ino/8), buf1);   // read inode block of ino
      ip = (INODE *)buf1 + (ino % 8);
  }
  if ((u16)ip->i_block[12]) // read indirect block into buf2, if any
     getblk((u16)ip->i_block[12], buf2);
  setes(0x1000);              // set ES to loading segment
  for (i=0; i<12; i++){     // load direct blocks
      getblk((u16)ip->i_block[i], 0);
      inces(); putc('*');   // show a * for each direct block loaded
  }
  if ((u16)ip->i_block[12]){ //load indirect blocks, if any
      up = (u32 *)buf2;
      while(*up++){
         getblk((u16)*up, 0);
         inces(); putc('.'); // show a . for each ind block loaded
       }
  }
  prints("ready to go?"); getc();
}
```

The booter's C code is fairly simple and straightforward. However, it is still worth pointing out the following programming techniques, which help reduce the booter size. First, if a booter needs string data, it is better to define them as string constants, e.g. name[0] = "boot", name[1] = "mtx", etc. String constants are allocated in the program's data area at compile-time. Only their addresses are used in the generated code. Second, on a FD the number of blocks is less than 1440. However, the block numbers in an inode are u32 long values. If we pass the block number as u32 in getblk() calls, the compiled (16-bit) code would have to push the long blk value as 16-bit items twice, which increase the code size. For this reason, the parameter blk in getblk() is declared as u16 but when calling getblk(), the long blk values are typecast to u16. The typecasting not only reduces the code size but also ensures getblk() to get the right parameters off the stack. Third, in the search() function, we need to compare a name string with the entry names in an EXT2 directory. Each entry name has name_len chars without an ending null byte, so it is not a string. In this case, strncmp() would not work since !strncmp("abcde","abcd", 4) is true. In order to compare the names, we need to extract the entry name's chars to make a string first, which require extra code. Instead, we simply replace the byte at name_len with a 0, which changes the entry name into a string for comparison. If name_len is a multiple of 4, the byte at name_len is actually the inode number of the next directory entry, which must be preserved. So we first save the byte and then restore it later. Finally, before changing ES to load the OS image, we read in the image's indirect blocks first while ES still points at the program's segment. When loading indirect blocks we simply dereference the indirect block numbers in the buffer area as *(u32 *). Without these techniques, it would be very difficult to write such a booter in C in 1024 bytes. Figure 3.15 shows the screen of booting MTX from a file system. In the figure, each asterisk is loading a direct block and each dot is loading an indirect block of the image file.

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 (1 KB) disk blocks. During booting, we prefer to load the image by blocks. As pointed out earlier, starting from segment 0x1000, loading 1 KB blocks will not cross any cylinder or 64 KB boundary. In

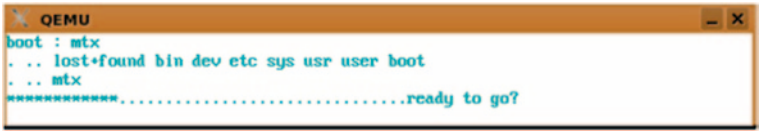


Fig. 3.15 Booting MTX from file system

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 64 KB 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 64 KB boundary. It would be very hard to write such a booter in 1 KB. 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 1 KB 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
  // serach 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
       inces();
     }
     //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)
        inces(); 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
}
```

Fig. 3.16 Booting linux from file system

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

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 Fig. 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;
    diskr();
}
```

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;
    diskr();
}
```

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 (1 MB) 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 1 MB 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 (64 KB) to 0x01000000 (16 MB), 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 (64 KB) in real mode memory to 0x01000000 (16 MB) in high memory.

```
C code:
         struct GDT gdt;     // define a gdt struct
         init_gdt(&dgt);     // 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 (4 KB 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 64 KB 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

```
stuct 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 Fig. 3.17.

The first sector of each extend partition is a local MBR. Each local MBR has

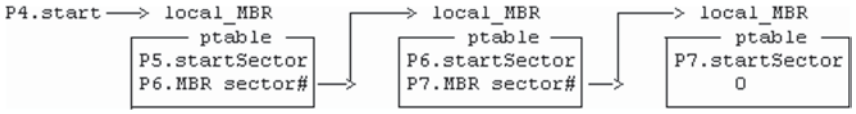


Fig. 3.17 Link list of extended partitions

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 8 GB. For disks larger than 8 GB 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
     gdblk = 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+gdblk, 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 cp2himem(), loading kernel image to 1 MB 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 4 MB. 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 cc
```

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

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

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 Chap. 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 1 MB and initrd to 32 MB. 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
        jmpi 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
        jmpi 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
        jmpi 0,  0x9020         ! jmpi 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 underlaying 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 4 KB. 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 contisuous 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
   }
}
```

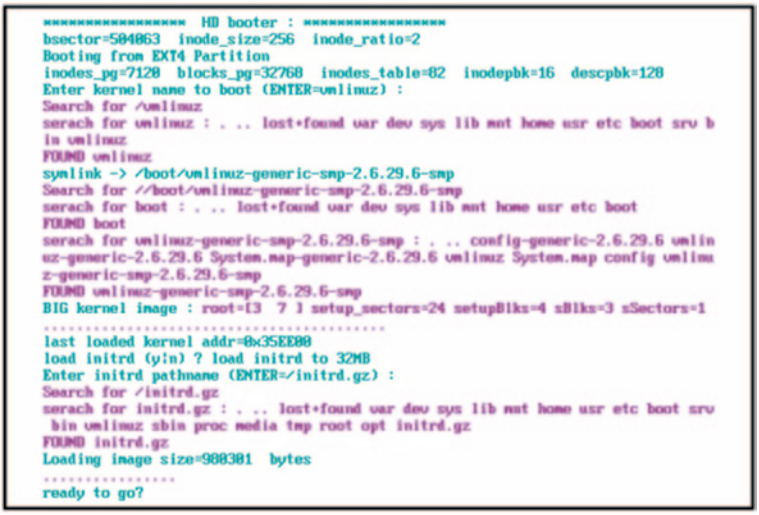


Fig. 3.18 Booting linux bzImage with initrd from EXT4 partition

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.

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 31 KB (the hd-booter size is about 10 KB). 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 vsupport 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 = 1.44 MB). 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 Chap. 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 Sect. 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_l_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 in 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 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
        jmpi 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);
     gdtp->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 (Syslinux 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 3.19 shows the booting screen of the iso-booter. It loads the Linux kernel to 1 MB and initrd.gz to 16 MB.

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-

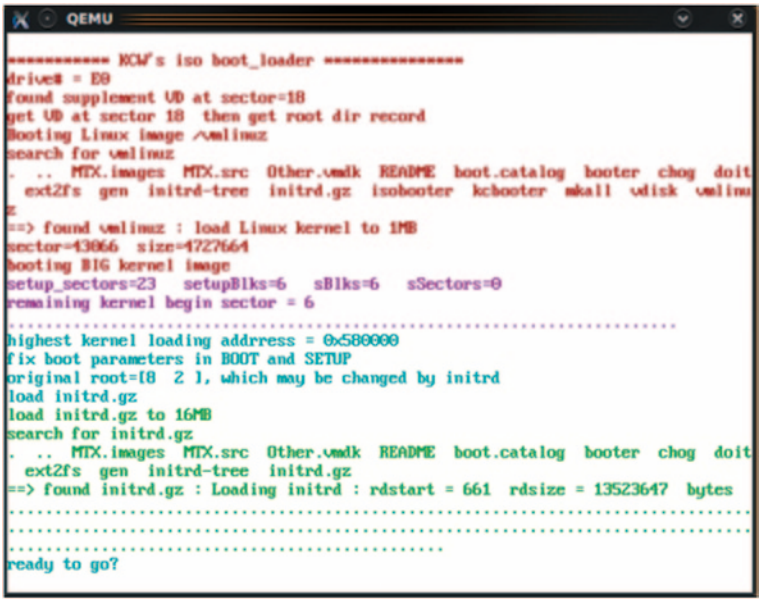


Fig. 3.19 CDOM iso-booter screen

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 8./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 1990’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, 32–64 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 s, 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 s 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
```

Fig. 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 1 KB 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 64 KB 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. Wrtie C code for the functions

getblk(u16 blk, char buf[1024])/putblk(u16 blk, char buf[1024])

which read/write a 1 KB 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.
11. Modify the hd-booter to accept input parameters. For example, when the booter starts, the user may enter an input line

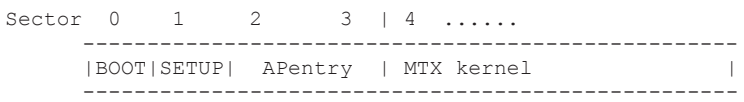
```
kernel=/boot/newvmlinuz initrd=/boot/initrd.gz root=/dev/sda7
```

where each parameter is of the form KEYWORD=value.

12. Modify the hd-booter to allow symbolic-link filenames for initrd.gz in the hd-booter.
13. The iso-booter does not handle symbolic-link filenames. Modify the C code to allow symbolic links.
14. Use the path table of an iso9660 file system to find the parent directory record.
15. USB booting: In some USB drives, a track may have less than 20 sectors. The hd-booter size is just over 10 KB. 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.

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



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.

17. 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

BusyBox: www.busybox.net, 2006

Cao, M., Bhattacharya, S, Tso, T., "Ext4: The Next Generation of Ext2/3 File system", IBM Linux Technology Center, 2007.

Comer, D.E., "Internetworking with TCP/IP: Principles, Protocols, and Architecture, 3/E", Prentice-Hall, 1995.

Comer, D.E., Stevens, D.L., "Internetworking With TCP/IP: Design, Implementation, and Internals, 3/E", Prentice-Hall, 1998.

GNU GRUB Project: www.gnu.org/software/grub/, 2010

Jones, M.T, "Linux initial RAM disk (initrd) overview", IBM developerworks, linux, Technical library, 2006

Slakware Linux: slackware.osuosl.org/slackware/README.initrd, 2013

Standard ECMA-119, Volume and File Structure of CDROM for Information Interchange,2nd edition, December 1987.

Stevens, C.E, Merkin, S. The "El Torito" Bootable CD-ROM Format Specification, Version 1.0, January, 1995

Syslinux project, www.syslinux.org, 2014