Chapter 3. Booting Operating Systems

Abstract: Chapter 3 provides a complete coverage on operating systems booting. It explains the booting principle and the booting sequence of various kinds of bootable devices. These include booting from floppy disk, hard disk partitions, CDROM and USB drives. Instead of writing a customized booter to boot up only MTX, it shows how to develop booter programs to boot up other operating systems, such as Linux, from a variety of bootable devices. In particular, it shows how to boot up generic Linux bzImage kernels with initial ramdisk support, and it demonstrates the booter programs by sample systems. It is shown that the hard disk and CDROM booters developed in this book are comparable to GRUB and isolinux in performance.

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 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 64KB 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 1KB below 0x08000. When execution starts, the actions of a booter are typically as follows.

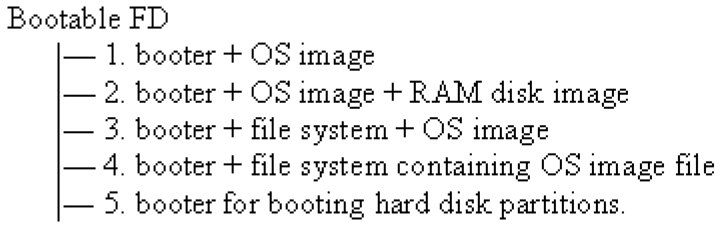
. Load the rest of the booter into memory and execute the complete booter.
. Find and load the operating system image into memory.
. Send CPU to execute the startup code of the OS kernel, which starts up the OS.

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 Figure 3.1. In the following, we shall describe the setup and booting sequence of each case.



```
Bootable FD
    |— 1. booter + OS image
    |— 2. booter + OS image + RAM disk image
    |— 3. booter + file system + OS image
    |— 4. booter + file system containing OS image file
    |— 5. booter for booting hard disk partitions.
```

Figure 3.1. Bootable FDs by Contents

(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 Figure 3.2.

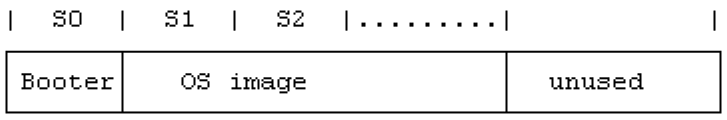


Figure 3.2 . A simple Bootable FD layout

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 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 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.

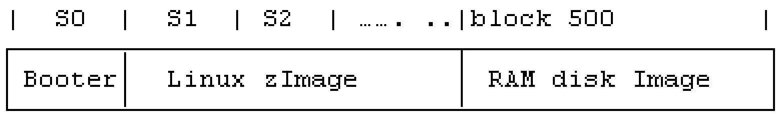


Figure 3.3. Bootable FD with RAM disk Image

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.

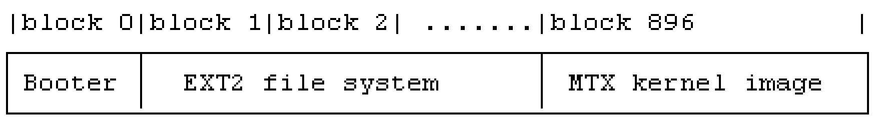


Figure 3.4. FD with EXT2 file system and MTX kernel

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, Figure 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 four16-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.

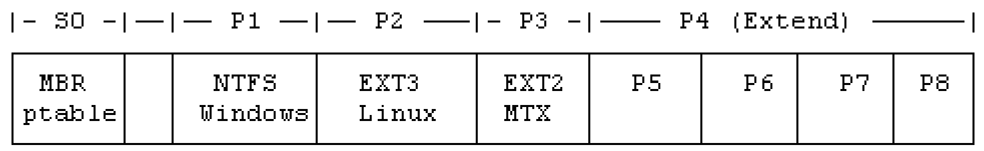


Figure 3.5. Hard Disk Partitions

(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. 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 Figure 3.6.

```
CD/DVD Booting
   |— emulation booting
   |          |— emulate FD  : install a bootable FD image
   |          |— emulate HD : install a (single-partition) HD image
   |— no-emulation booting: install an arbitrary bootable image
```

Figure 3.6. CD/DVD Boot Options

(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/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] 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. Multi-stage 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 operating 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 Figure 3.7.

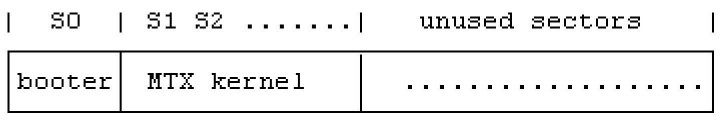


Figure 3.7. MTX Boot Disk layout

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, 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.

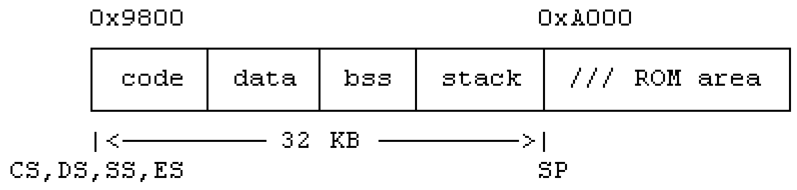


Figure 3.8. Run-time image of booter

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.
```

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.

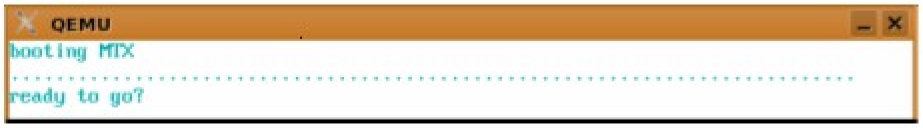
```
Linear:  0 ....... 17 18 .........35 36 ...... 53 54 ..... 71
         ____________________________________________________________
sector: |S0 ....  S17|S0 .....    S17|S0 ..... S17|S0 ...   S17|
  head: | ── head=0 ── | ───head=1 ─── | ───head=0 ───|── head = 1 ─|
   cyl: |<──────── cyl = 0 ────────>|<────── cyl = 1 ───────>| etc.
```

Figure 3.9. Linear Sector and CHS addressing

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.

    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 Section 3.3.5. Figure 3.10 shows the booting screen of the MTX.sector booter, in which each dot represents loading a disk sector.



```
booting MTX
..............................................................................
ready to go?
```

Figure 3.10. Booting Screen of MTX.sector Booter

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 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 Figure 3.11.

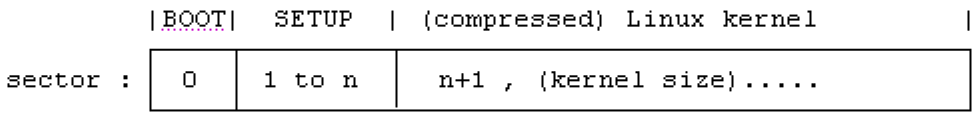


Figure 3.11. Bootable Linux Image

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=zImage2.4 of=/dev/fd0 bs=512 seek=1 conv=notrunc
```

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

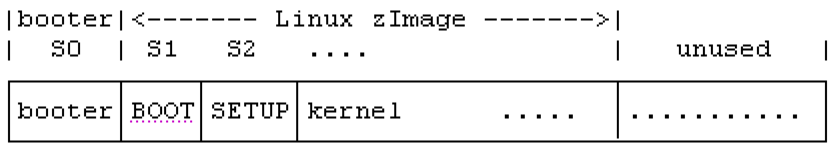

|booter|<------- Linux zImage ------->|
| S0 | S1 S2 .... | unused |
booter
BOOT
SETUP
kernel .....
....

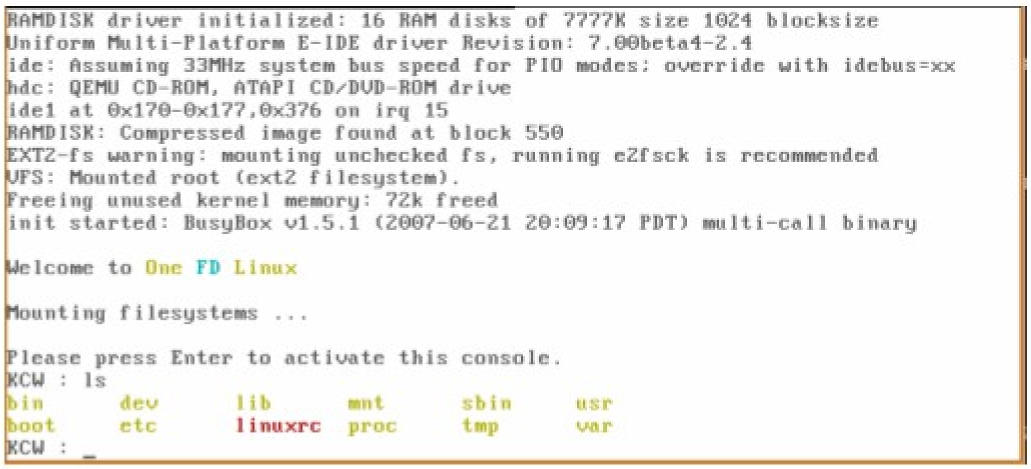
Figure 3.12. Linux Boot Disk layout

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\rrady to go?"); getc();
}
```

The booting screen of the Linux zImage booter is similar to Figure 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 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.



```
RAMDISK driver initialized: 16 RAM disks of 7777K size 1024 blocksize
Uniform Multi-Platform E-IDE driver Revision: 7.00beta4-2.4
ide: Assuming 33MHz system bus speed for PIO modes: override with idebus=xx
hdc: QEMU CD-ROM, ATAPI CD/DVD-ROM drive
ide1 at 0x170-0x177,0x376 on irq 15
RAMDISK: Compressed image found at block 550
EXT2-fs warning: mounting unchecked fs, running e2fsck is recommended
VFS: Mounted root (ext2 filesystem).
Freeing unused kernel memory: 72k freed
init started: BusyBox v1.5.1 (2007-06-21 20:09:17 PDT) multi-call binary

Welcome to One FD Linux

Mounting filesystems ...

Please press Enter to activate this console.
KCW : ls
bin      dev      lib      mnt      sbin     usr
boot     etc      linuxrc  proc     tmp      var
KCW : _
```

Figure 3.13. Single-FD Linux running on Ramdisk

3.3.5. Fast FD Loading Schemes

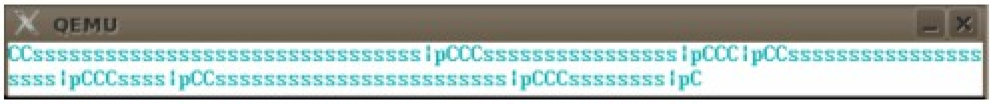
The above FD booters load OS images one sector at a time. For small OS images, such as the MTX kernel, this works fine. For large OS images like Linux, it would be too slow to be acceptable. A faster loading scheme is more desirable. When boot a Linux zImage, logically and ideally only two loading operations 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 64KB boundary problem, which says that when loading FD sectors the real memory address cannot cross any 64KB 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 64KB 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 64KB 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 1KB blocks would also work. In order not to cross both cylinder and 64KB 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 cylinder, each | is crossing a 64KB boundary and each p is loading the remaining sectors of a cylinder.



```
CCssssssssssssssssssssssssssssssss|pCCCssssssssssssssss|pCCC|pCCssssssssssssssss
ssss|pCCCssss|pCCssssssssssssssssssssssss|pCCCssssssss|pC
```

Figure 3.14. Booting Linux by Loading Cylinders

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 Section 2.8.2 of Chapter 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.

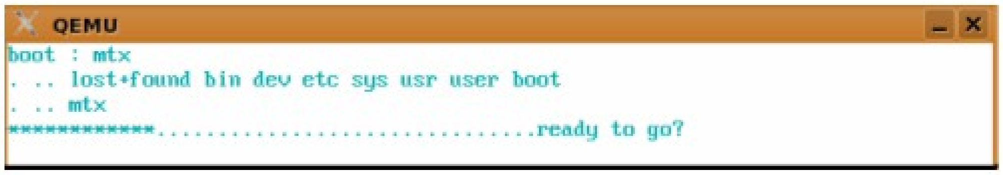
When such a booter starts, it must be able to access the file system on the boot disk, which means loading disk blocks into the booter program’s memory area. In order to do this, we add a parameter, buf, to the assembly function, readfd(char *buf), where buf is the address of a 1KB memory area in the 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 1KB 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.



```
boot : mtx
. .. lost+found bin dev etc sys usr user boot
. .. mtx
************..............................ready to go?
```

Figure 3.15. Booting MTX from File System