

Bootable USB Device
Aaron Arthurs
Don Hayes
Bailey Hendrickson

Table of Contents

Abstract	1
Introduction	1
Background Information	3
Configuration for Gentoo Linux	6
Preparing a Knoppix Distribution For USB	8
Diskless Boot Procedure for RedHat 8 Linux With TFTP	11
Biography of Each Team Member	13

Abstract

The popularity of USB mass storage devices (e.g. external hard drives, pen/thumb drives, external CD/zip) motivates the need for USB boot capabilities. Most modern computers and their BIOS do not support the ability to directly boot a USB mass storage device; therefore, third-party software such as boot loaders and USB drivers is needed to aid the boot process. Due to the nature of such software, only image-based kernels can boot from a USB device. The kernel images themselves must be available from various sources such as a local, non-USB drive or from a network drive via TFTP.

Introduction

In order for a personal computer to boot to a USB (Universal Service Bus) mass storage device, at least one of the two conditions must exist. One, the BIOS (Basic Input-Output System) supports booting off USB, which directly loads the boot loader on the USB device. Two, if the BIOS doesn't support booting to the USB device, the boot loader, modified kernel, and initial ram-disk image (discussed later), must be stored in a form that the BIOS can directly read, e.g. CD, floppy, or internal hard drive.

For this project, it is assumed that the BIOS will not directly read the USB device; therefore, USB support must be provided via software stored on BIOS-readable storage. The method used is provide the kernel and initial ram-disk images on a BIOS-readable disk. The kernel and ram-disk images, in this case, are suited to boot a Linux kernel. The following section is a brief primer on how booting with the needed images works.

Background Information

To boot a Linux kernel, a boot loader is needed. The boot loader, once loaded by the BIOS, will load the kernel and initial ram-disk (also called initdisk or initrd) images into memory, which will, in turn, be booted by the boot loader (i.e. the boot loader passes control to the loaded images); moreover, the boot loader may read the images from the internal hard disk, a CD, a floppy, or a TFTP (Trivial File Transfer Protocol) server. Once the Linux kernel is loaded by the images, the root file system ('/') is mounted; in turn, programs can execute and additional kernel modules can be loaded for additional functionality.

However, to mount the root file system, which is stored on a USB device, some conditions must be met; the kernel must load the appropriate drivers in order to access the device containing the root file system (e.g. USB drivers). Furthermore, the root file system may be encrypted and necessitate prompting the user for a password. For a USB mass storage device, there are various solutions for integrating the needed drivers. One, the USB drivers can be compiled into the kernel. The disadvantages to this approach are that the USB drivers may not function well with other drivers and that the kernel can become very large. Another possibility is to provide different kernels, each of which contain some USB drivers. This solution also has its problems as it can dramatically increase the number of kernels needed; this problem is more apparent with optimized kernels (e.g., Pentium, Athlon optimized).

Another approach is to load the USB drivers as modules; however, this leads to the problem where the USB driver modules are stored on the USB device. The solution is then to create an initial ram-disk, the method used for

running user-space programs before the root file system is mounted. With the initial ram-disk, the Linux kernel will load a small file system into a ram-disk and execute certain programs before mounting the real root file system.

The boot loader (LILO, GRUB, syslinux etc.) loads the initial ram-disk; any storage media the BIOS can enumerate will be passed to the boot loader, requiring the initial ram-disk (and the kernel image) to be stored on one of the enumerated storage media. The main point is that third-party drivers are not needed for a storage device if the BIOS can interpret it.

The boot loader loads the kernel and the initial ram-disk images into memory and starts the kernel, telling it that an 'initrd' is present and where in memory the kernel will find it. If the initrd was compressed (which is typically the case), the kernel decompresses and mounts it as a temporary root file system. After having done so, a special program named linuxrc, contained in the initrd, is executed. This program is able to perform all tasks necessary to enable the kernel to mount the real root file system, which is where the USB drivers are loaded. When linuxrc terminates, the (temporary) initrd is unmounted and booting continues by mounting the real root file system, which is now mountable because of the loaded USB drivers. The mounting of the initrd and the execution of linuxrc may thus be viewed as a short intermezzo during the normal boot procedure.

The program linuxrc has only to meet the following requirements: It has to have the name 'linuxrc' and it has to be in the root directory of the initrd. Besides that, the kernel has to be able to execute it. This means that linuxrc may well be linked dynamically, but then of course the shared libraries have to be completely available under /lib in the initrd. Additionally, linuxrc may also be a shell script,

necessitating the need of a shell, available under `/bin`. In short, the `initrd` must contain a minimal Linux system that allows the program `linuxrc` to be executed. `Linuxrc` is run with root privileges.

As soon as `linuxrc` terminates, the `initrd` is unmounted and discarded and booting continues normally with the kernel mounting the real root file system. What gets mounted as the root file system may be changed by `linuxrc`. To do so, `linuxrc` only has to mount the `/proc` file system and write the value of the real root file system in numerical form to `/proc/sys/kernel/real-root-dev`. Once the initial `linuxrc` program is complete the `/sbin/init` is executed and the console is initialized from the USB device and booting continues normally.

Configuration for Gentoo Linux

The first necessary step is for a kernel to be compiled with USB drivers compiled in. When using a 2.6.x kernel, this is accomplished by setting the following drivers to be compiled in when using 'make menuconfig'.

- Device Drivers->USB support->EHCI (USB 2.0) Enhanced
 - HCI stands for 'Host Controller Interface'
- Device Drivers->USB support->OHCI Open
- Device Drivers->USB support->UHCI(for Intel and via chipsets) Universal
- Device Drivers->USB support->USB Mass Storage

The device drivers are compiled into the kernel (not as modules). A new kernel and supporting modules for the other drivers must be built and installed.

This is accomplished via the following commands:

```
# cd /usr/src/linux
# make bzImage
# make modules
# make modules_install
```

Changing to the /usr/src/linux directory is necessary so the make utility can find the needed targets. make bzImage builds a compressed kernel image suitable for use in booting on the i386 architecture. The make modules and make modules_install build the appropriate modules for the kernel to use and installs them. An initial ram-disk must also be built; this is accomplished via the following:

```
# mkinitrd initrd-2.6.5 2.6.5
```

Note that this is for users using the 2.6.5 kernel (the version used on the Gentoo test bed) if using a different kernel version, replace 2.6.5 with the appropriate version number. This command generates an initial ram-disk with the filename

'initrd-2.6.5'; any filename could have been used, but appending the kernel version to 'initrd' is the clearest.

The next step is to move those files to a location suitable for booting. On the Gentoo test bed system, this location is the /boot partition, partitioned on the disk as /dev/hda1. First the /boot partition is mounted so it can be accessed.

```
# mount /dev/hda1 /boot
# cp /usr/src/linux/arch/i386/boot/bzImage /boot/kernel-2.6.5
# cp /usr/src/linux/initrd-2.6.5 /boot/initrd-2.6.5
```

The kernel is then copied to the boot partition (changing the name to make it clear). The initrd file generated earlier is likewise copied.

The final step is to write a configuration file so the boot loader knows which files to use. The following is the pertinent contents of 'grub.conf':

```
#grub bootloader configuration file
default 0
timeout 17

splash=verbose
splashimage=(hd0,0)/grub/splash.xpm.gz

title=Gentoo linux-2.6.5
root (hd0,0)
kernel /kernel-2.6.5 root=/dev/ram0 real_root=/dev/hda3 vga=791
initrd /initrd-2.6.5

title=Suse 9.0
root (hd0,0)
kernel /kernel-2.6.5 root=/dev/ram0 real_root=/dev/hdb3 vga=791
initrd /initrd-2.6.5

title=usb kernel-hd0(hda) root-sda3 linux-2.6.5
root (hd0,0)
kernel /kernel-2.6.5 root=/dev/ram0 real_root=/dev/sda3 vga=791
initrd /initrd-2.6.5

title=usb hd2-sda3 linux-2.6.5
root (hd2,0)
kernel /kernel-2.6.5 root=/dev/ram0 real_root=/dev/sda3 /vga=791
initrd /initrd-2.6.5
```

Preparing a Knoppix Distribution For USB

To demonstrate the feasibility of booting Linux off a small USB flash-drive an image based version of Linux known as Knoppix was used. The specific distribution of Knoppix is 'Damn Small Linux', which is only 50 Megabytes in size and has a crude windowing environment. The reason an image-based version of Linux is ideal with a flash-drive is due to the fact that the file system can be the more compatible FAT (File Allocation Table) as opposed to the Linux only EXT2 file system. This allows the user to be able to boot off the drive and still use it as auxiliary storage if using a windows machine. Also the fact that DSL is Knoppix based allows for more auto-configuration at boot, which is also needed to allow users added hardware support and flexibility. The following is a procedure on how to modify the initial ram-disk to allow it to install the USB drivers and boot Knoppix off a flash-drive.

1. Download and decompress DSL or other version of Knoppix. It might be easier just to mount the CD image and copy all of the files over without having to burn a CD. Keep in mind the amount of space available on drive. For the author's installation a 256MB flash-drive was used. For clarity it is assumed that the Knoppix distribution is uncompressed and in the folder /temp/
2. We need to modify the initial ram-disk which is called miniroot.gz and is stored in the boot-image at /temp/KNOPPIX/boot.img. To use this image mount it with the following command 'mount -o loop boot.img /mnt/temp1'.
3. Now that we have the miniroot.gz file we need to decompress it and mount it like we did the boot-image.

```
gunzip -d /mnt/temp1/miniroot.gz /temp/  
# /mnt/temp1 is mounted boot-image  
mount -o loop /temp/miniroot /mnt/temp2
```
4. Now that mini-root is mounted and available we need to modify the linuxrc file so the USB drivers are loaded before the SCSI drivers and the boot process must be paused for a small amount of time because it typically take a few seconds before the USB device is available for mounting. The linuxrc file should be edited by adding the following lines after the SCSI drivers are loaded. An actual linuxrc file is included on the CD. For clarity, the paths in the following are relative to the initial ram-disk. Later we have to add the USB drivers so that linuxrc file will have access to

them. Since we're editing a file already in an image we have to save linuxrc in another temp directory and copy it over.

```
# Load USB drivers before SCSI so that the USB device will be /dev/sda1
insmod -f /modules/usb/ehci-hcd.o
insmod -f /modules/usb/usbcore.o
insmod -f /modules/usb/usb-uhci.o
insmod -f /modules/usb/uhci.o
insmod -f /modules/usb/usb-ohci.o
insmod -f /modules/usb/usb-storage.o
# USB drivers are loaded but we need to sleep to work out kinks
ash -c "sleeping momentarily"
```

5. Now since Knoppix is originally a CD based distribution and we're booting it off a USB device we need to trick it so that it will run correctly off the flash-drive. The following lines changed in the linuxrc file are sufficient.

```
# Mount USB flash-drive to /cdrom
FOUND_KNOPPIX="/dev/sda1"
mount -t vfat /dev/sda1 /cdrom
```

6. Now we're done editing the linuxrc file but we are referencing files that aren't originally located in the miniroot image. This means we have to add the files to the new miniroot. Suitable USB drivers can be found with most recent versions of Linux. In this case we mount the Knoppix image file that houses the real root filesystem. A set of USB drivers can be found on this image. The commands to do so look like.

```
Insmod cloop file=/temp/knoppix/KNOPPIX &&/
mount -t iso9660 /dev/cloop /cdrom
cd /cdrom/lib/modules/$KERNEL_VERSION/kernel/drivers/usb/
cp ehci-hcd.o /mnt/temp2/ #/mnt/temp2 is where minroot is mounted
cp uhci.o /mnt/temp2/
cp usb-ohci.o /mnt/temp2/
cp usb-storage.o /mnt/temp2/
cp usb-uhci.o /mnt/temp2/
cp usbcore.o /mnt/temp2/
cd ~
```

7. Now that the mini-root image has the USB drivers and modified linuxrc file so it can use them we can unmount it and recompress it.

```
umount /mnt/temp2
rm /temp/miniroot.gz # Delete old miniroot.gz image
gzip /temp/miniroot
```

8. Now we have to prepare the USB drive and install the needed files to boot. The easiest way to do it is to copy all of the files from the boot disk image. Then copy the entire /temp directory which includes the updated miniroot.gz and vmlinuz kernel which are necessary to boot. Syslinux can be installed on the flash-drive, which would enable it to boot normally if the host computer had native USB support. Syslinux works well for booting Linux systems off of FAT partitions, which makes especially useful here.

```
mkfs.vfat /dev/sda1 # Assuming /dev/sda1 is correct could be different
mount /dev/sda1 /mnt/usb
cp /mnt/temp1/* /mnt/usb/
cp -R /temp/*
umount /mnt/usb
syslinux /dev/sda1 # Optional: Installs Syslinux to drive
```

9. Now the USB flash-drive is prepared and the two necessary files for booting are created in the root directory of the drive (vmlinuz and miniroot.gz).

Diskless Boot Procedure for RedHat 8 Linux With TFTP

The RedHat 8 installer comes with USB support, which allows for ease of installation to a USB mass storage device. Also, the needed images themselves come with USB capabilities, allowing for the Linux kernel to boot from a USB device.

In order to utilize a TFTP server for the purpose of booting RedHat 8 from a USB device, GRUB is used in conjunction with local PXE capabilities and a DHCP server. Depending upon the model of the computer, its BIOS must be configured to boot from a network. When the BIOS attempts to boot from network, it first looks for a DHCP server in order to be assigned an IP address; furthermore, the DHCP server will point to the TFTP server as well as the boot file to load (in this case, the file containing the GRUB boot loader). The client computer will then load the boot file.

Below is a detailed procedure for preparing and using GRUB along with the TFTP server.

1. Obtain and un-tar GRUB.
2. Configure and build GRUB as follow.
 - a. `cd <grub directory>`
 - b. `./configure --enable-diskless --enable-<your network card>`
 - c. `make`
3. Copy stage2/pxegrub to the TFTP server's root directory.
4. On the DHCP server, specify the TFTP server's IP address and the boot file 'pxegrub'.
5. Optionally, create a GRUB configuration file under the directory <TFTP ROOT>/boot/grub/'. 'pxegrub' will look for a file named 'menu.lst' under that directory.
6. At this point, the client computer should be able to boot from 'pxegrub'.
7. As for the kernel and initial ram-disk images, copy 'vmlinuz' and 'initrd.img' from the boot disk to the TFTP server. To make a boot disk, execute the following.
 - a. `uname -a` (this shows the kernel's version number)
 - b. `mkbootdisk --device /dev/fd0 <kernel's version number>`
8. Once the client computer boots into 'pxegrub', execute the following.
 - a. `root (nd)`

- b. kernel /vmlinuz root=/dev/sda2 (assuming default Linux installation)
- c. initrd /initrd.img
- d. boot

Biography of Each Team Member

Aaron Arthurs:

Aaron Arthurs is a senior in Computer Engineering at the University of Arkansas, Fayetteville. He entered the University of Arkansas in the fall of 2000 after graduating from the Fayetteville High School. He has a strong interest in graphics hardware design, animation, and artificial intelligence; his hobbies include hiking and game design. Aaron will enter graduate school at the University of Arkansas in fall of 2004. Also, he has accepted a job in website design with TCBI Inc., located in the same town, which will be in the summer of 2004. Aaron was born in Fayetteville, AR, and attended K-12 schools in the same town. Aaron hopes to own his own company in computer graphics. Aaron can be contacted at ajarthurs@tcbi.dyndns.biz.

Don Hayes:

Don Hayes is a senior in Computer Engineering at the University of Arkansas. He entered the University of Arkansas in the fall of 2000 after graduating from Farmington High School in Farmington, AR. He has a strong interest in software design and game design and his hobbies include designing strategy and war games and playing various games. Don is currently looking for a job in the Northwest Arkansas area. Don was born in Fayetteville, Arkansas and attended Farmington Schools in Farmington, Arkansas. Don hopes someday to be a professional game designer. Don can be contacted at quartic_@hotmail.com (with underscore) or 479-267-2097.

Bailey Hendrickson:

Bailey Hendrickson is a senior in Computer Engineering at the University of Arkansas at Fayetteville. He transferred to the University of Arkansas from Hendrix College in 2001. He graduated from the Arkansas School for Mathematics and Sciences in 1999. He has an interest in distributed computing and software design with various languages. He is in the process of applying to graduate school at several major universities. Bailey was born in Conway, Arkansas and attended public and private schools during his academic career there. Bailey hopes to get a lucrative career in software development. He can be contacted at jhendri@uark.edu.