ELJonline:
BOEL, Part 2: Kernel Configuration and Booting

Brian Elliott Finley

Brian's Own Embedded Linux, or BOEL as it is affectionately known, was created as part of the SystemImager Project. SystemImager is a tool used for, well, imaging systems and distributing live updates. Part of the imaging process involves booting a client machine in such a way that its hard disks are not in use and are therefore available for manipulation. The SystemImager autoinstall client does this by using a root filesystem that exists entirely in memory or an initrd (initial RAM disk). This initrd and the kernel that goes with it are the embedded Linux system we call BOEL.

This is the second article of a two-part series. In the first article, we went through the process of creating our initrd and discussed issues such as libraries, shells, filesystems and inodes. Much of our focus is on creating a system as small as possible -- the complete BOEL system has to fit on a single 1.44MB floppy. In this article we go through the process of tying a kernel to our initrd and making the system bootable from a variety of media.

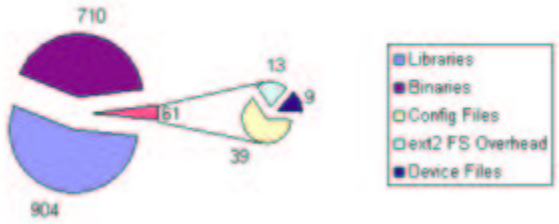

Figure 1. Uncompressed RAM Disk Components (in Kilobytes)

Compiling the Kernel

One nice thing about compiling a kernel for an embedded system is that you typically have a very limited set of hardware to support. In the case of BOEL, the only types of hardware we have to support are NICs (Network Interface Cards), disk drives and disk drive controllers. This makes it much easier to pare down the drivers needed for our kernel. It also allows us to use a non-modular kernel, which in addition to simplifying the boot process, also simplifies the development process. Even as such, it can still take a lot of time to whittle down a kernel to exactly what is needed and nothing more. But it's worth making that effort. Every additional driver takes up valuable space on our system, so we only want to include the drivers that we need.

This means we can leave out things like USB, sound, Ham radio support, etc. However, we do have to be sure to include support for things like an initrd (we have to be able to mount the thing for our root filesystem), loopback devices (the initrd gets put on a loopback device before the kernel mounts it), NICs, TCP/IP, IDE, SCSI and the filesystems that we support. If you want to have a look at exactly what we include in our standard kernel, you can take a look at this kernel .config file.

Once you've made your driver selections, you must compile your kernel. Funny thing is, each time you compile it, your resultant kernel may have a slightly different size. I usually will do several compiles until I'm convinced I've gotten the smallest one possible.

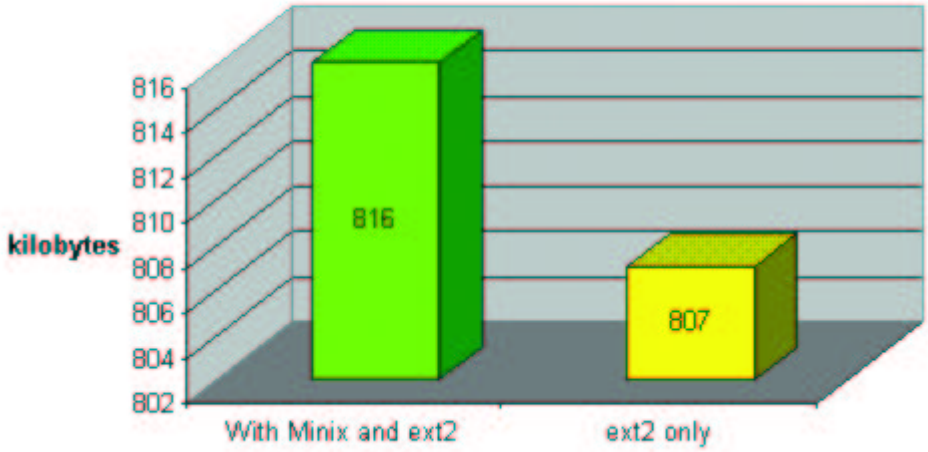

Figure 2. Kernel Sizes

Now that we have our kernel ready and our initrd as prepared in the first article, we need to configure the system for booting. In the case of BOEL, we need to be able to boot from floppy, CD-ROM, a hard drive or the network. I'll describe how we make each of these happen using the same kernel and initrd.

Booting the System

First, a bit about booting in general. When booting a system that uses an initrd, the actual steps the computer goes through are not necessarily intuitive. On a Linux system that has its root filesystem on a hard disk partition (with no initrd), the boot loader simply loads the kernel and tells it where to find its root filesystem. When the kernel finishes initializing the hardware, it happily mounts the root filesystem, and you're off and running.

With an initial RAM disk, the initrd must be decompressed and placed into memory before the kernel is loaded. Otherwise, there would be no root filesystem available for the kernel to mount when it finishes its initialization. Here's a summary of the steps taken by a boot loader when using an initrd:

1. Decompress the initial RAM disk (usually stored as initrd.gz).
2. As it is being decompressed, toss it up into memory.
3. Toss the kernel up into memory (without overwriting the RAM disk), telling it the memory address where the RAM disk starts, the device it's supposed to use to mount the RAM disk

(/dev/ram0), and kick the kernel to start it spinning through the processor.

4. Once the kernel finishes initializing, it mounts the RAM disk as its root filesystem as if nothing special happened, using the device location that it was told by the boot loader.

H. Peter Anvin has created a package, called SYSLINUX, that contains some utilities that make it really quite easy to get a system with an initrd to boot from a floppy, a CD-ROM or from the network. In recent releases, there are three separate tools for the three different boot methods mentioned above: SYSLINUX, ISOLINUX and PXELINUX. These tools can use a configuration file (syslinux.cfg, isolinux.cfg or pxelinux.cfg) to specify boot options and a messages file (messages.txt) to give the person booting the system important messages, like "This is the SystemImager autoinstall system. If you do not want your disks repartitioned and freshly formatted, power off now!"

Booting from a Floppy

To make our floppy bootable, we first must format the floppy with the FAT (File Allocation Table) filesystem. FAT is the only filesystem that is readable by SYSLINUX. After formatting the floppy, we copy over the following files: initrd.gz, kernel, syslinux.cfg and messages.txt. In our case, the syslinux.cfg file contains the following entries:

```
DEFAULT kernel
APPEND vga=extended load_ramdisk=1 prompt_ramdisk=0
       initrd=initrd.gz root=/dev/ram rw
DISPLAY message.txt
PROMPT 1
TIMEOUT 50
```

The contents of the messages.txt file do not affect the booting of the system, so I won't list them here. Once these files have been copied to the floppy, we unmount the floppy and run the command `syslinux -s /dev/fd0`. That's it! Our embedded Linux system can now boot from floppy!

Booting from a CD-ROM

When BOEL was first created, ISOLINUX wasn't around, so we used SYSLINUX for our CD-ROM booting as well as for floppies. Therefore, making a CD-ROM bootable is very similar to making a floppy bootable. As a matter of fact, the method we currently use is to create a bootable floppy, then use that bootable floppy as the basis for an ISO image. But to make things faster, we actually employ a file that we mount using the loopback device instead of a floppy diskette:

```
dd if=/dev/zero of=/tmp/diskette.img bs=1k count=2880
mkdosfs /tmp/diskette.img
syslinux -s /tmp/diskette.img
mkdir -p /tmp/mount_dir/
mount -t msdos -o loop /tmp/diskette.img
/tmp/mount_dir/
cp initrd.gz kernel syslinux.cfg message.txt
/tmp/mount_dir/
umount /tmp/mount_dir/
```

And finally, we issue the commands to create our El Torito-style bootable CD-ROM:

```
mkdir -p /tmp/iso_tmp_dir/boot/
cp /tmp/diskette.img /tmp/iso_tmp_dir/boot/
cd /tmp/iso_tmp_dir/
mkisofs -b boot/diskette.img -c boot/boot.catalog
-o /tmp/CD_Image.iso .
```

Now let me comment on a couple of things you may have noticed in the commands above. We use a 2.88MB floppy image instead of the 1.44MB floppy size you might expect; 2.88MB is a valid floppy size, and it gives us a little more breathing room. Since this will only take up a fraction of the capacity of the CD, why not? (Honestly, I can't remember why we decided to go with 2.88 instead of 1.44.)

You also may have noticed that SYSLINUX was run prior to placing the kernel on the floppy image. SYSLINUX, like GRUB but unlike LILO, is actually able to read the filesystem where the kernel resides. This means that it can find and read the configuration file, which always has the same name. From the configuration file it can determine the filenames of the kernel and initrd. It can then access the kernel and initrd as normal files that it needs to work with. This is in contrast to LILO, which must pre-record the physical location of your kernel on the disk.

The entire bootable CD-ROM and floppy creation process is automated in SystemImager with the mkautoinstallcd and mkautoinstalldiskette commands.

Booting from the Network

For booting from the network, we use PXELINUX. PXE (pronounced "pixie") stands for pre-execution environment. With PXELINUX, the process is very similar to using SYSLINUX. We can even use the same configuration and message files, but the configuration file must be named differently. I'll explain why in just a bit.

To boot a machine using PXELINUX, you must first configure another machine as a DHCP (Dynamic Host Configuration Protocol) and TFTP (Trivial File Transfer Protocol) server. SystemImager comes with two utilities that make this easy by walking you through the process: mkdhcpserver and mkbootserver. In addition to the basic entries needed for handing out IP addresses, mkdhcpserver also adds information that will be used by clients booting with PXE. Most notably, it specifies the location of the PXELINUX binary from the perspective of a PXE-enabled network card. In our case that means /pxelinux.bin.

The mkbootserver command will make sure that your TFTP dæmon is installed and enabled. It also will copy the PXELINUX files to the right location so they can be served up by the TFTP dæmon.

Once you have a machine set up as a DHCP and TFTP server, you will need to tell your client to boot from its network card. This is usually done by changing your client's BIOS settings, but also is available as a "press this key now to activate" during boot time option on some systems. If you enter your BIOS to change this setting, you will need to look for a bit about "Boot Order". This should list all of the bootable devices in your computer and will include any PXE-enabled NICs that you may have.

Now when you power on your computer, instead of asking the hard drive for boot information, it will ask the NIC. The NIC will make a DHCP request that will be answered by your DHCP server. The DHCP server will provide the NIC with the IP address settings it should use and the location of the

PXELINUX binary. The NIC will then use TFTP to retrieve the PXELINUX binary from the TFTP server. The PXELINUX binary is then run by the client system as a boot loader. If we were using SYSLINUX, this is the point where it would read its configuration file from a FAT partition. PXELINUX, on the other hand, retrieves it from across the network and reads it from the data stream.

I mentioned earlier that the configuration file could be the same as with SYSLINUX but must be named differently. As a matter of fact, in our case, we've written the configuration file, configured the TFTP server and located the kernel and initrd.gz files so that we can use the same exact configuration file for booting from floppy, CD and network.

And with regards to configuration files, PXELINUX gives you some nice flexibility by allowing you to specify a different configuration file for each machine or a single file for a group of machines. It doesn't know enough to ask for a file by hostname, but it does know its IP address. So it first requests a configuration file named with the IP address that it is using. The catch is that it uses the hexadecimal representation of this IP address. If it can't find a configuration file with that name, it lops off one character from the end and tries again until it runs out of characters in the filename. At that point it makes one last desperate try and asks for a file named "default". It also copies down the messages.txt file, but there's nothing exciting to say about that.

Why try all those variations? Because it allows you to specify a configuration file based on an individual computer or based on subnets of increasing sizes. We just use the default file by default.

Once PXELINUX gets its hands on the configuration file, it retrieves the kernel and RAM disk as specified in the file, also via TFTP. The kernel and RAM disk filenames are relative to the root of what is presented by the TFTP server. At this point, the boot process proceeds in exactly the same way as with SYSLINUX -- the kernel and initial RAM disk are tossed up into memory, and off we go!

There is one network booting trick I should mention. It's kind of a hack and I consider it dangerous for production use, but in special circumstances it can be useful. Most modern BIOSes are smart enough to try each boot device in succession if the previous one fails. What this means is that you can configure your system to boot from the network first, and then from the hard drive if that fails. Assuming that you usually want your systems to boot from the hard drive, you just cripple part of your network boot configuration by way of the DHCP or TFTP servers until you need it to work. The advantage is that your machines are already configured in a way that you can remotely deal with a failed hard drive (depending on how it fails). The disadvantage is that anyone who can plug a DHCP/TFTP server on to your network can do malicious things with your systems.

Booting from a Hard Drive or Anything that Looks Like a Hard Drive

SystemImager's updateclient utility provides an option that allows you to connect to a remote machine that needs to be re-installed and remotely prepares it to be "autoinstalled". Those of you familiar with SystemImager will recognize this as "that wonderful -a option". When you run `updateclient -autoinstall -server <servername>`, it contacts the imageserver specified, pulls down BOEL's kernel and initrd and configures the system to load these at boot time.

After the kernel and initrd.gz files have been copied to /boot on the hard drive, updateclient adds an entry to LILO's configuration file /etc/lilo.conf. That entry will look something like this:

```
image=/boot/kernel
```

```
label=systemimager
initrd=/boot/initrd.gz
read-write
# appending root=/dev/ram0 ensures that the kernel
# will not try to run the nonexistent /linuxrc
# script
append="LAST_ROOT=/dev/sda3 root=/dev/ram0
        load_ramdisk=1 prompt_ramdisk=0"
```

After adding this entry, updateclient will run the command `lilo -d 50 -D systemimager`. This tells LILO to load the systemimager entry with a five-second delay on the very next boot, but after that one deviant boot session, to return to booting the default kernel. This allows you to bring your system back to its previous useable state if your autoinstall client fails to initiate the install for some reason.

You also may notice three interesting things about the lilo.conf entry above. One is that it actually has comments to tell you what's going on. This is a theme that we have tried to carry on throughout BOEL and the SystemImager distribution as a whole. When you are making so many small decisions where each affects the behavior of another piece of the puzzle several steps down the line, it is incredibly important to be able to recall why you did something that may look a little odd six months from now.

Skipping the second item for a minute to provide continuity (noncontinuity for the sake of continuity), the third item of interest is the LAST_ROOT= append parameter. Append parameters are a way to give the kernel configuration details. You can think of it as setting variables for the kernel. And you hard-core kernel hackers will be noting that LAST_ROOT is not a valid append parameter -- that's because I made it up! It's a variable that is used by SystemImager to identify the root filesystem that was in use prior to starting an autoinstall by way of booting from the hard drive. This allows BOEL to look for a SystemImager configuration file, local.cfg, that can be used to control some of the autoinstall behavior, such as your client's IP address, your imageserver's IP address and your desired image name, among others.

No, I didn't have to hack the kernel to use this variable. As a matter of fact, the kernel doesn't use it at all! But, it is visible in user space by means of the /proc/cmdline kernel table. Take a look to see what yours has in it: `cat /proc/cmdline`. Not much there, huh? Well, even if you passed your kernel "a whole mess of options", as Bill Cosby would say, only the first few at the very beginning show up here. And that's why LAST_ROOT is the very first append parameter. Lest you be concerned that you have stuff in /proc/cmdline but have no append parameters specified, it's probably because your boot loader automatically passes your kernel some of the things it needs in order to boot properly. You can do a `man bootparam` to get the lowdown on all the official append parameters.

Finally, the second item of interest, which leads into the next section, is the comments themselves. Why would appending root=/dev/ram0 ensure that the kernel would not try to run the nonexistent /linuxrc script? Well, at the time I added that comment, there was a bug/feature in the Linux kernel that caused /linuxrc always to be executed when using an initial RAM disk -- except when you used /dev/ram0 as the root device. You can actually choose to use any of the available RAM devices as your root device, but I didn't want to run /linuxrc. I wanted to go straight to /sbin/init. Nowadays the kernel has been fixed, and you can simply use the boot parameter init=/linuxrc if you still want to use a script or other custom executable instead of going straight to /sbin/init.

Hacking DHCP to Send and Receive Nonstandard Information

So why bother with dhclient on the RAM disk if space is so important? Why not just use the "IP: kernel-level configuration support" option in the kernel? Because, we want to do mischievous things with DHCP, things like creating our own options to send nonstandard data. And we want the ability to use these options to control the behavior of our embedded Linux system, not just affect boot-related parameters.

Now, if you've ever taken a look at the dhcp-options man page, you know that there's a whole mess of options that you can specify in your dhcpd.conf file. And if you were looking closely, you may have noticed that you can also make up your own. That's right, you can pass any information you like to your DHCP clients by adding a line to your DHCP server like so:

```
option option-133
"stuff-what-I-want-to-send-to-my-clients";
```

Now the problem is, what can my client do with it? If you are using either dhcpcd (DHCP client dæmon) or pump (not an acronym, can you believe it?), and they receive an option they don't recognize, they just drop it on the floor and stomp on it a few times. Both have a very limited set of fixed options that they understand. Of course, we could hack the code to make them recognize the nonstandard options we want to send, but do we really want to do that every time a new version of the code comes out?

Although dhcpcd and pump seem to be the two most popular DHCP clients, there is a third called dhclient. **dhclient** seems to be rarely used, even though it is part of the same package as the DHCP server software made by ISC that is the de facto standard. In contrast to dhcpcd and pump, dhclient is able to accept any and all of the options that its sibling server software is able to send -- even the ones you make up.

dhclient uses a configuration file (/etc/dhclient.conf) and a script (/etc/dhclient-script). This gives us a lot of room for customizations that will hopefully survive code changes. And it turns out that the options specified at the server simply show up as variables while dhclient script is running. So if we wanted to capture the information passed from the server as option-133, we can access that data in the variable $new_option_133. For BOEL, we add commands to the dhclient-script that stores these values so that they can be easily retrieved by our embedded system -- as a sourceable dot-style file. Here is an example line that we add to this script:

```
echo "SSH_DOWNLOAD_URL=$new_option_208"
>> /tmp/dhcp_variables.txt
```

BOEL is distributed as part of SystemImager and is available in source form here and `make initrd` will produce these two components of BOEL. Thanks to Dann Frazier and the entire SystemImager development team.

***About the author: Brian Elliott Finley** has specialized in operating systems, system architecture and system integration since 1990. In 1993 he fell in love with Linux.*

SystemImager is a project begun in 1998 as a way to update multiple Solaris machines across the US.
