March 29, 2020

BOOTING LINUX OVER HTTP

—

A couple years ago, one of my friends gave me a big pile of little Dell FX160 thin clients, which are cute little computers which have low power Atom 230 processors in them with the ability to support 3GB of RAM. Being thin clients means they were originally meant to be diskless nodes that could boot a remote desktop application to essentially act as remote graphical consoles to applications running on a beefier server somewhere else.

That being said, they're great as low power Linux boxes, and I've been deploying them in various projects over the years when I need a Linux box somewhere but want/need something a little more substantial than a Raspberry Pi.

The one big problem with them is that they didn't come with the 2.5" hard disk bracket, so I needed to source those drive sled kits on eBay to add more storage than the 1GB embedded SATA drive they all came with. Which is nominally fine; I bought a few of the kits for about $10 a piece, and for that to be the only expense to be able to deploy a 1TB 2.5" drive somewhere has been handy a few times.

But it always left me thinking about what I could do with the original 1GB drive in these things. Obviously, with enough effort and hand wringing, you can get Linux installed on a 1G partition, but that feels like it's been done before, and these are thin clients! They're meant to depend on the network to boot!

Fast forward to this year, and thanks to one of their network engineers hearing my interview for On the Metal, I've been working with Gandi.net to help deploy one of their DNS anycast nodes in Fremont as part of the Fremont Cabal Internet Exchange. The thing is, how they designed their anycast DNS nodes is awesome! They have a 10,000 foot view blog post about it, but the tl;dr is that they don't deploy their remote DNS nodes with an OS image on them. Each server gets deployed with a USB key plugged into them with a custom build of iPXE, which gives the server enough smarts to, over authenticated HTTPS, download the OS image for their central servers and run the service entirely from RAM.

Operationally, this is awesome because it means that when they want to update software on one of their anycast nodes, they can build the new image in advance on their provisioning server centrally, and just tell the server to reboot. When it reboots, it automatically downloads the new image from the provisioning servers, and you're up to date. If something goes terribly wrong and the OS on a node becomes unresponsive? Open a remote hands ticket with the data center "please power cycle our server" and the iPXE ROM will once again download a fresh copy of the OS image to run in RAM.

Granted, they've got all sorts of awesome extra engineering involved in their system; cryptographic authentication of their boot images, local SSDs so while the OS is stateless, their nodes don't need to perform an entire DNS zone transfer from scratch every time it reboots, etc, etc. Which is all well and good, but this iPXE netbooting an entire OS image over the wide Internet using HTTP is just the sort of kind-of-silly, kind-of-awesome sort of project I've been looking to do with these thin clients I've got sitting around in my apartment.

Understanding The Boot Process

This left me with a few problems:

1. The Gandi blog post regarding their DNS system was a 10,000 foot view conceptual overview, so they rightfully-so glossed over some of the technical specifics that weren't important to their blog post's message but really important for actually making it work.
2. I have been blissfully ignorant up until now of most of the mechanics involved with Linux booting in the gap between "The BIOS runs the bootloader" and "The Linux kernel is running with your Init server running as PID 1 and your fstab mounted"
3. I'm trying to do something exceedingly weird here, where there are no additional file systems to mount while the system is booting. There's plenty of guides available on booting Linux with an NFS or iSCSI root file system, but I'm looking at even less than that; I want the entire system just running from local RAM.

So before talking about what I ended up with, let's talk about the journey and what I had to learn about the boot process on Linux.

On a typical traditional Linux host, when you power it on, the local BIOS has enough smarts to find local disks with boot sectors, and read that first sector from the disk and execute it in RAM. That small piece of machine code then has enough smarts to load a more sophisticated bootloader like GRUB from somewhere close on the disk, which then has enough smarts to do more complicated things like load a Linux kernel and init RAM disk to boot Linux, or give the user a user interface to select which Linux kernel to boot, etc. One of the reasons why many Unix systems had a separate /boot partition was because this chainloader between the BIOS and the full running kernel couldn't mount more sophisticated file systems so needed a smaller and simpler partition for just the bare minimum boot files needed to get the kernel running.

The kernel file plus init RAM disk (often called initrd) are the two files Linux really requires to boot, and the part where my understanding was lacking. Granted, my understanding is still pretty lacking, but the main insight I gained was that the initrd file is a packed SVR4 archive of the bare minimum of files that the Linux kernel needs to then go and mount the real root file system and switch to it to have a fully running system. These SVR4 archives can be created using the "cpio" command as the "newc" file format, and the Linux kernel is smart enough to decompress it using gzip before mounting the archive, so we can gzip the initrd file to save bandwidth when ultimately booting the system.

(Related aside; there's many different pathways from the BIOS to having the kernel and initrd files in RAM. One of the most popular "net booting" processes, which I have used quite a bit in the past, is PXE booting, where the BIOS boot ROM in the network card itself has *juuuust* enough smarts to send out a DHCP request for a lease which includes a TFTP server and file path for a file on that TFTP server as DHCP options, and the PXE ROM downloads this file and runs it. This file is usually pxelinux.0, which I think is another chainloader which then downloads the kernel and initrd files from the same TFTP server, and you're off to the races.)

The missing piece for me inside the initrd file is that the kernel immediately runs a shell script in the root of the filesystem named "/init". This shell script is what switches the root file system over to whatever you specified in your /etc/fstab file, and ultimately at the very end of the /init script is where it "exec /sbin/init" to replace itself with the regular init daemon which you're used to being PID 1 and being the parent of every other process on the system.

I had never seen this /init script before, which is understandable because it's normally not included in your actual "/" root file system! It's only included in the initrd archive's "/" file system (which you can actually unpack yourself using gunzip and cpio), and disappears when it remounts the actual root and exec's /sbin/init... So since I want to run Linux entirely from RAM, "all" I need to do is figure out how to create my own initrd file, generate one that is not a bare minimum to mount another file system but everything I need to run my application in Linux, and figure out a simpler /init script to package with it which doesn't need to mount any local volumes but only needs to mount all the required virtual file systems (like /proc, /sys, and /dev) and exec the real /sbin/init to start the rest of the system.

Generating My Own Initrd File

So the first step in this puzzle for me is figuring out how to generate my own initrd file including the ENTIRE contents of a Linux install instead of just the bare minimum to get it started. And to generate that initrd archive, I first need to create a minimal root file system that I can configure to do what I want to then pack as the initrd file we'll be booting.

Thankfully, Debian has some really good documentation on using their debootstrap tool to start with an empty folder on your computer and end up with a minimal system. The first section of that documentation talks about partitioning the disk you're installing Debian on, but we just need the file system, so I skipped that part and went straight to running debootstrap in an empty directory.

```
$ sudo debootstrap buster /home/kenneth/tmp/rootfs http://ftp.us.debian.org/debian/
```

Remember that there's plenty of Debian mirrors, so feel free to pick a closer one off their list.

Once debootstrap is done building the basic image, from a terminal we can jump into the new Linux system using chroot, which doesn't really boot this system, but jump the terminal into it like it was the root of the currently running system, so you can interact with it *like* it's running. This lets us edit config files like /etc/network/interfaces, apt install needed packages, etc etc. Pretty much just following the rest of the Debian debootstrap guide and then also doing the configuration work needed to set up whatever the system should actually be doing. (things like setting a root password, installing ssh, configuring network interfaces, etc etc)

```
$ LANG=C.UTF-8 sudo chroot /home/kenneth/tmp/rootfs /bin/bash
```

Since we're not installing this system on an actual disk, we don't need to worry about installing the GRUB or LILO bootloader like the guide says, but I did install the Linux kernel package since it was the easiest way to grab a built Linux kernel to pair with the final initrd file we're creating. Apt install linux-image-amd64 and copy that vmlinuz file out of the .../boot/ directory in the new filesystem to somewhere handy.

The next step is to place the much simpler /init script in this new file system, so when the kernel loads this entire folder as its initrd we don't go off and try and mount other file systems or anything. This is the part where my friend at Gandi.net was SUPER helpful, since trying to figure out each of the various virtual file systems that still need to be mounted on my own only yielded me a lot of kernel panics.

So huge thanks to Arthur for giving me this chunk of shell code! Copy it into the root of the freshly debootstrapped system and mark it executable (chmod +x)

Source for init:

```
#!/bin/sh
# Kenneth Finnegan, 2020
# https://blog.thelifeofkenneth.com/
# Huge thanks to Gandi.net for most of this code

set -x
set -e

# Create the mount points for all of the virtual file systems which don't
#  actually map to disks, but are views into the kernel
[ -d /dev ] || mkdir -m 0755 /dev
[ -d /root ] || mkdir -m 0700 /root
[ -d /sys ] || mkdir /sys
[ -d /proc ] || mkdir /proc
[ -d /tmp ] || mkdir /tmp
mkdir -p /var/lock || true

# Mount the required virtual file systems
mount -t sysfs -o nodev,noexec,nosuid sysfs /sys
mount -t proc -o nodev,noexec,nosuid proc /proc

tmpfs_size=10240k
if ! mount -t devtmpfs -o size=$tmpfs_size,mode=0755 udev /dev; then
        echo "W: devtmpfs not available, falling back to tmpfs for /dev"
        mount -t tmpfs -o size=$tmpfs_size,mode=0755 udev /dev
        [ -e /dev/console ] || mknod -m 0600 /dev/console c 5 1
        [ -e /dev/null ] || mknod /dev/null c 1 3
fi
unset tmpfs_size

mkdir /dev/pts
mount -t devpts -o noexec,nosuid,gid=5,mode=0620 devpts /dev/pts || true
mount -t tmpfs -o "nosuid,size=20%,mode=0755" tmpfs /run

# Set dmesg to private if you want
echo 1 > /proc/sys/kernel/dmesg_restrict

# Replace ourselves with the actual init daemon which will handle starting every other daemon
exec /sbin/init
```

At this point, we're ready to pack this filesystem into an initrd archive and give it a shot. To create the archive, I followed this guide, which boils down to passing cpio a list of all the file names, and then piping the output of cpio to gzip to compress the image.

```
$ cd /home/kenneth/tmp/rootfs
$ sudo find . | sudo cpio -H newc -o | gzip -9 -n >~/www/initrd
```

At this point, you should have this initrd file which is a few hundred MB compressed, and the vmlinuz file (vmlinuz being a compressed version of the usual vmlinux kernel file!) which you grabbed out of the /boot directory, and that *should* be everything you need for booting Linux on its own. Place both of those files on a handy HTTP server to be downloaded by the client later.

Netbooting This Linux Image

Given the initrd and kernel images, the next step is to somehow get the target system to actually load and boot these files. Aside from what I'm talking about here of using HTTP, you can use any of the more traditional booting methods like putting these files on some local storage media and installing GRUB, or using the PXE boot ROM in your computer's network interface to download these files from a TFTP server, etc.

TFTP would probably be pretty cute since many computers can support it stock, but that depends on your target system being on a subnet with a DHCP server that can hand out the right DHCP options to tell it where to look for the TFTP server. I didn't want to depend on DHCP, and I wanted to use HTTP, so I instead opted to use iPXE, which is a much more sophisticated boot ROM than the typical PXE ROMs you get.

It is possible to directly install iPXE on the firmware flash of NICs, but that's often challenging and hardware specific, and a good point that Arthur pointed out was that since they boot iPXE from USB, if for some reason they need to swap the iPXE image remotely, it's *MUCH* easier to mail a USB flash drive and ask them to replace it than to try and walk someone else through how to reflash the firmware on a NIC over the phone... I'm not going to be using a USB drive, since these thin clients happen to have convenient 1GB SSDs in them already, but it's the same image. Instead of dd'ing the ipxe.usb image onto a flash drive, I just temporarily booted Linux on the thin clients and dd'ed the ipxe ROM onto the internal /dev/sda.

The stock iPXE image is pretty generic, and like a normal PXE ROM sends out a DHCP request for a local network boot image to download. This isn't what we want here, so we're definitely going to need to build our own iPXE binary in the end, but I started with the stock ROM because it allows you to hit control-B during the boot process and interactively poke at the iPXE command line, and manually step through the entire process of configuring the network, downloading the Linux kernel, downloading the initrd file, and booting them.

So before building my own custom ROM, I burned iPXE onto a USB flash drive and poked at the iPXE console with the following commands on my apartment network:

```
dhcp
kernel http://example.com/vmlinux1
initrd http://example.com/init1
boot
```

And that was enough to start iterating on my initrd file to get it to what I wanted. Since I was still doing this in my apartment which has a DHCP server, I was able to ask iPXE to automatically configure the network with the "dhcp" command, then download a kernel and initrd file, and then finally boot with the two files it just downloaded.

So at this point, I was able to boot the built Linux image interactively from the iPXE console, and had a fully running Linux system in RAM, which was kind of awesome, but I wanted to fully automate the iPXE booting process, which means I need to build a custom image with an embedded "iPXE script" which is essentially just a list of commands for iPXE to run to configure the network interface, download the boot files, and boot.

iPXE Boot Script:

```
#!gpxe
ifopen net0
set net0/ip 192.0.2.100
set net0/netmask 255.255.255.0
set net0/gateway 192.0.2.1
set net0/dns 192.0.2.1

echo Configuring network...
sleep 3

kernel http://example.com/vmlinux1
initrd http://example.com/init1

echo And away we go!
boot
```

ipxe.script hosted with ❤ by **GitHub** **view raw**

So given that script, we follow the iPXE instructions to download their source using git, install their build dependencies (which I apparently already had on my system from past projects, so good luck...), and the key step is that when performing the final build, we pass make the path to our iPXE boot script file to embed it in the image as what to run.

```
$ cd ~/src/ipxe/src
$ make EMBEDDED_IMAGE=./bootscript bin/ipxe.usb
```

And at this point in the ipxe/src/bin folder is the built image of ipxe.usb which has our custom boot script embedded in it! Since the internal SATA disk is close enough to a USB drive, from a booting perspective, that's the variant of ROM I'm using.

So given this custom iPXE ROM, I manually booted a live Linux image on the thin client, used dd to write the ROM to /dev/sda which is the internal 1G SSD, and the box is ready to go!

Now, when I power on the box, the BIOS sees that the internal 1G SSD is bootable, so it boots that, which is iPXE, which runs the embedded script we handed it, which configures the network interface, downloads our custom initrd file and the Linux kernel from my HTTP server, and boots those. Linux then unpacks our initrd file, and runs the /init script embedded in that, which just mounts the virtual file systems like /proc/, /sys/, and /dev, and then *doesn't* try and mount any other local file system, and finally our /init/ script exec's /sbin/init, which in the case of Debian happens to be systemd, and we're got a fully running system in RAM!

Video of generally what that looks like:

So once again, thanks to Arthur from Gandi.net for the original idea and gentle nudges in the right direction when I got stuck.

Of course, the next thing to do is start playing "disk space golf" with the OS image to see how small I can make the initrd file, since the smaller the initrd file, the more RAM that is left over for running the application in the end! And actually doing something useful with one of these boxes running iPXE... a topic for another blog post.

Update: One thing to note is that this documentation is for the minimum viable "booting Linux over HTTP". iPXE does support crypto such as HTTPS, client TLS certificates for client authentication, and code signing. More details can be found in their documentation.

Share

POPULAR POSTS

November 15, 2017

CREATING AN AUTONOMOUS SYSTEM FOR FUN AND PROFIT

May 05, 2021

UNLOCKING THIRD PARTY TRANSCEIVERS ON OLDER ARISTA SWITCHES

Share

Kenneth Finnegan

VISIT PROFILE

Archive

Pr
M
Sc

Easily Sea
Project M

Workforc