LinuxBoot: Booting FreeBSD from Linux

BY WARNER LOSH

After maintaining the FreeBSD UEFI boot loader for years at Netflix, our hardware designer asked me to improve system booting resilience. After failing to find a solution with the stock UEFI firmware, I looked for alternatives. Most alternative firmwares boot the OS using a stripped-down Linux kernel, so I implemented LinuxBoot support for FreeBSD. I'll describe how I did that, but first I'll explain what LinuxBoot is, where it came from, what it does, and where it fits in. In a second article, I'll describe the nuts and bolts of creating LinuxBoot firmware images that boot FreeBSD.

Over the past five years a new variation on booting has matured and become popular. Spurred on by a desire to reduce the attack surface for the boot phase, Linux now boots Linux. This seemingly awkward arrangement has advantages over traditional UEFI booting. Some new embedded environments offer only LinuxBoot. Facebook, Google, IBM, Microsoft, and Apple have supported these efforts to improve security, reduce boot complexity, and provide a common platform largely independent of underlying architecture. FreeBSD 14.0 offers preliminary support for booting FreeBSD/aarch64 and FreeBSD/amd64 with a new variation of /boot/loader, called "loader.kboot," which uses LinuxBoot.

(A note on terminology: I use "aarch64" and "amd64" because they are more visually distinct than "arm64" and "amd64," which are easily confused.)

Three major themes have led us to the point where LinuxBoot is gaining popularity.

How We Got Here

Three major themes have led us to the point where LinuxBoot is gaining popularity: initial simplicity, uncontrolled growth, and a desire to return to a simpler time. All three themes have contributed to the complex booting ecosystems that have sprung up on both x86 and embedded systems. LinuxBoot attempts to simplify those ecosystems somewhat, though having the entire Linux kernel involved usually doesn't make one immediately think of simplicity.

In the days before the IBM PC, most systems either required a bootstrap to be entered manually or the automated boot ROMs were simple enough to load a boot sector and it would load the rest of the system with it. This process of using simple loaders to load progressively more complex loaders is called "bootstrapping" the system, from the old saying "pulling yourself up by your own bootstraps." In time, this was shortened to "booting" the system.

In 1982, IBM's release of the IBM PC only slightly improved on prior systems by providing both the bootstrapping code in its ROMs and other basic I/O. It called these services the BIOS from the term used by CP/M systems that preceded it. This is where we get the term "BIOS" and why there's some confusion surrounding whether it means "the firmware used

to bootstrap the system" or whether it only refers to the pre-UEFI style of booting on the PC platform. Partisans of both stripes are sure they are right, but few recognize the history behind this ambiguity. So, I'll use "CSM" or "CSM booting" to refer to this style of bootstrap. The UEFI Standard uses the term "CSM" to describe legacy booting, and it is unambiguous.

In time, all kinds of additional features were added to CSM booting: a partition table for disks, the MP Table for processor configuration, APM for power management, SMBIOS for metadata about the system, run time services for PCI, PXE network booting, and ACPI to unify many of the prior features. The interfaces for these services were tied to x86-specific mechanisms. The system evolved into a very complex ecosystem, riddled with quick hacks, special cases, and subtly different interpretations.

When Intel designed the IA-64 CPU architecture in the late 1990s, it quickly discovered that the revolutionary architecture couldn't use the vast majority of techniques from the CSM ecosystem. The whole booting ecosystem had to be replaced. This was the genesis of Unified Extensible Firmware Interface (UEFI) booting. At first, it was just on the Intel x86 (re-branded IA-32) and IA-64 architectures. During the early 2000s, UEFI firmware slowly displaced the legacy systems on Intel x86 systems, oftentimes being able to do either the old CSM booting or the new UEFI booting. By 2006, Intel had released, via its TianoCore project, the EDK2 open source development kit for UEFI firmware creation, prompting even more OEMs to adopt UEFI.

In time, all kinds of additional features were added to CSM booting.

Also, during the 1990s and 2000s, another booting ecosystem was evolving for embedded systems. Initially, the embedded space had dozens of different bootloaders, all subtly different in their interface to later stages of booting. Mangus Damm and Wolfgang Denk created Das U-Boot in 1999, first for PowerPC but later ARM, MIPS, and other architectures. U-Boot started out small and simple, more flexible than its competitors, and relatively easy to extend because it was open source. It quickly became the universal bootloader because of its simplicity, support, and rich feature set. It set the standard for booting, and drove many features in Linux, including flattened device tree (FDT) support. This boot system had nothing in common with either CSM or UEFI. It was so easy to use that all its competitors have faded into obscurity. Where are redboot, eCos, CFE, yaboot, or YAMON today? Footnotes on a Wikipedia page at best.

In 2011, ARM introduced aarch64, its 64-bit version of the ARM platform. Both U-Boot and EDK2 vied for dominance to bootstrap the system, while FDT and ACPI vied for dominance to enumerate system devices. Low-end, embedded systems tended to use U-Boot with FDT, while higher-end, server-class systems used UEFI and ACPI. Eventually, UEFI booting started to win out, especially when U-Boot started to ship a minimal UEFI implementation sufficient to boot Linux via UEFI. ACPI and FDT merged (you can specify ACPI nodes now with FDT properties). And through it all, EDK2/UEFI grew more and more complex to support SecureBoot, iSCSI, more NICs, RAM disk support, initramfs support, and too many other features to list.

And that doesn't even count all the bootloaders using a stripped-down version of the Linux kernel, such as coreboot, slimboot, LinuxBIOS, and others, some of which I'll describe below. Nor does it begin to describe the variation between commercial BIOSes. By 2017, Google decided to do something about this situation and started project NERF to simplify this mess and harden security. The name stands for Non-Extensible Reduced Firmware, as opposed to UEFI's Unified Extensible Firmware Interface. It is also slang from computer gaming for a change that downgrades the power or influence of a game element to achieve a better balance or improve enjoyment of the game. These efforts would later become LinuxBoot.

Linux Booting

Booting Linux with Linux has a very long history, but space allows only a brief summary. In the mid-1990s, when the kexec(2) family of system calls was added to Linux, it was used to increase uptime and/or reliability of servers and embedded systems. Ron Minnich and Eric Biederman started LinuxBIOS at Los Alamos in the 1990s to use a Linux kernel in the Firmware to boot the system. This evolved into coreboot, used by Chromebooks and several open platform laptops. In the process, coreboot became modular to allow binary blobs alongside the open source components because CPU manufacturers have resisted opening up the early processor initialization code, providing only binary blobs to both open and closed source firmware creators. EDK2, U-Boot and closed-source firmware have also developed a modular system to allow these binary blobs to exist alongside other components.

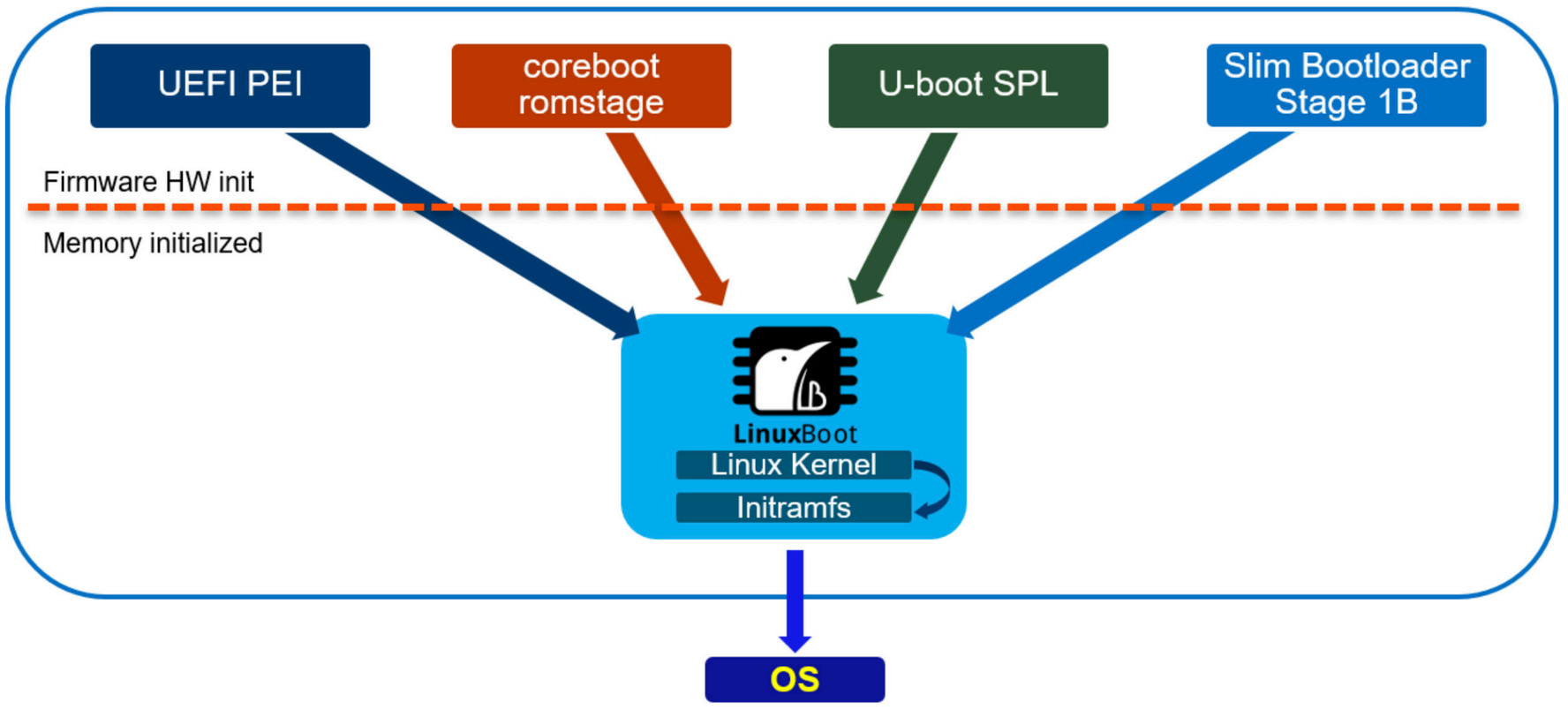
They wanted to create a common framework using widely deployed and reviewed code.

Project NERF Becomes LinuxBoot

Google's NERF project, headed by Ron Minnich, evolved into the LinuxBoot project: a series of scripts that help create firmware images that boot the final OS with the Linux kernel. The project had several bigger-picture goals behind it, however. Google wanted to create an open source firmware where every component was freely available. They wanted to simplify the UEFI booting environment, which they felt had grown too complex with too many potential security vulnerabilities, by replacing it with the hardened Linux kernel. They wanted to create a common framework using widely deployed and reviewed code; minimize the unavoidable, binary-only, non-source portions of the bootloader; unify booting for the ARM and other embedded systems as much as possible; eliminate redundant code, speeding the boot; and create a more modular and customizable boot experience than traditional firmware or even EDK2 provided. They wanted a reproducible build, which ensures anybody can run the exact same binary whether they download it or build it themselves.

The result was a modular system that supported many bootloaders. The earliest stages of initializing the CPU were handled by CPU-specific and bootloader-specific code. LinuxBoot defined what parts of the system were initialized there and which parts were deferred to the Linux kernel. This setup allowed CPU vendors to continue to ship binary-only blobs that initialized the low-level clocks, memory controllers, auxiliary cores, etc. that a modern

CPU needs to become functional. EDK2, coreboot, U-Boot, and slim boot all support these protocols, so the Linux kernel boots with all of them, without special code for any of them. LinuxBoot also provided u-root, a ramfs builder written in go for finding and loading the final and a few other tools to manipulate firmware images. I'll discuss these tools and how to use them in the second part of this article.



Source: https://www.linuxboot.org

While not completely successful at replacing the entire bootloader with Linux, LinuxBoot minimized the amount retained. For example, with UEFI, only the Pre-EFI Initialization (PEI) phase initializing the processor, caches, and RAM, and UEFI's runtime services remained. LinuxBoot eliminated all of the thinly tested UEFI DEX drivers. The Linux kernel takes over with the memory and base hardware initialized, but without any of the other things that a more traditional firmware might initialize, like resources for PCI devices.

In addition to better security and more control over the firmware, LinuxBoot uses the well-tested, highly reviewed Linux drivers required for Linux to run on the platform. With LinuxBoot, SOC vendors and system integrators can optimize their time to market by writing drivers only for Linux. UEFI DEX drivers need not be created at all. Programmers with Linux driver skills are much easier to find than those who can write a UEFI DEX driver. The Linux kernel has been audited by thousands of researchers, compared to relatively few who have studied the EDK2 UEFI code base. These advantages, however, require other operating systems supporting UEFI to adapt. Their UEFI bootloaders do not work with the sliver of UEFI that remains. This means that to boot at all on these systems, an OS must create a new loader to support LinuxBoot.

Some simpler operating systems boot with LinuxBoot using the basic ELF loading that Linux's kexec-tools package has to offer. Very old versions of BSD and Plan9 have been booted this way. FreeBSD/powerpc, which runs on a processor from a simpler time with a well-defined OpenFirmware interface, also loads this way. Windows, however, cannot boot this way, and the LinuxBoot community is researching ways around it. FreeBSD/amd64 and FreeBSD/aarch64 cannot boot this way either.

FreeBSD kernels for amd64 and aarch64 require metadata available only to the bootloader. On amd64, the bootloader sets the system into long mode after capturing information about the system's memory layout and other data, which can only be accessed before

entering long mode. The kernel relies on this data and cannot operate without it. On both amd64 and aarch64, the bootloader has to inform the kernel of the address of the UEFI system table and other system data. The bootloader tunes the kernel by setting "tuneables." The loader pre-loads dynamic kernel modules and passes things like initial entropy, UUID, etc. into the kernel. None of this specialized knowledge is present in the kexec-tools that can only load an ELF binary and jump to its start address.

FreeBSD and LinuxBoot

FreeBSD's history with booting via Linux goes back over a decade. In 2010, the FreeBSD PS/3 port used the PS/3's "another OS" option to start. Nathan Whitehorn, a FreeBSD developer, added the necessary glue to the FreeBSD bootloader to set up memory for FreeBSD's kernel. He created a small Linux binary similar to Ubuntu's kboot from their PS/3 support package. This Linux binary was statically linked and included the few system calls needed to read the FreeBSD kernel off the PS/3 disk. It included a small libc (similar to mucl or glibc) and command line parsing support. However, the structure of its sources assumed only PowerPC.

While working at Netflix, I began experimenting in 2020 to see how hard it would be to boot FreeBSD with Linux. Netflix runs a large fleet of servers installed throughout the world. Through years of constant refinement, Netflix created a very robust system that corrects common problems automatically. Even after these refinements, booting issues caused too many costly RMAs. Experiments using UEFI scripting to improve boot-time reliability provided only marginal improvements. Because flash drives contained the scripts, only a few trivial cases improved. Flash drives can fail read-only, confusing both the UEFI firmware and the scripts into doing the wrong thing. As long as the scripts remained on the drives, progress was impossible.

Netflix began experimenting in 2020 to see how hard it would be to boot FreeBSD with Linux.

LinuxBoot offered an attractive alternative to UEFI because it resided inside the firmware on the motherboard, eliminating the components most prone to failure. Netflix wanted me to create a fail-safe environment that could phone home status information about the machine, reprovision the machine using surviving NVMe drives, and provide a flexible platform to enable remote debugging, diagnostic images, etc.

I had several goals with booting FreeBSD from Linux:

1. It had to be built within the FreeBSD build system.
2. It had to provide full access to host resources.
3. It had to boot with a stock kernel (if possible).
4. It had to use the UEFI boot interface (i386 CMS booting, and arm U-Boot binary booting would not be supported).
5. It had to run as init/PID 1.
6. It also had to run well when called from shell scripts to support booting different kinds of images not necessarily based on FreeBSD.

Getting FreeBSD booting from Linux on modern architectures like amd64 and aarch64 required several changes to the relatively modest PS/3 kboot base. The loader needed four types of changes: refactoring the existing kboot following the MI/MD model, expanding

support for accessing host resources, refactoring UEFI boot code to be used by both the UEFI loader.efi and the LinuxBoot loader.kboot, and retiring technical debt within the bootloader.

MI/MD Changes

Several areas needed the classic MI/MD split where common MI code interfaces with per-architecture MD code that implements a common API. Linux has a much larger difference in system calls between architectures than FreeBSD. Program startup needs slightly different assembler glue between architectures. Different linker scripts are needed. The loader metadata, while mostly similar, has architectural differences. Finally, the handoff from the Linux kexec reboot vector to the kernel differs. I'll cover these last two below in the Refactoring UEFI Booting section.

The first three of these changes are needed to create Linux binaries. To create static binaries, I wrote the C runtime support that provides the glue between the Linux kernel handoff and a more traditional main routine. I wrote a bit of per-architecture assembly, coupled with a standard startup routine that calls main. To accomplish this a standard C interface to the system calls allowed the MD part of FreeBSD's mini libc for Linux to be small. I created a small amount of per-architecture assembler for system calls. I added a framework for Linux's per-architecture ABI differences, the largest being in the termios interface. This reflects Linux's complicated history of binary compatibility. A per-architecture linker script produces a Linux ELF binary. These elements combine to make loader.kboot, Linux ELF binary. The new libsa drivers (see below) interface to this libc.

Accessing Host Resources

The original loader.kboot code accessed some host resources, but it was incomplete. I wanted to boot either off of a raw device, or via a kernel or loader residing in the filesystem of the host system. The bootloader has always supported a number of different ways to specify where files come from but before refactoring, adding new one was hard. With some changes to refactor the existing code, I added the ability to access any block device from its Linux name. The name "/dev/sda4:/boot/loader" reads the file /boot/loader that's within the fourth partition on the sda disk, for example. In addition, "lsdev" now lists all of the eligible Linux block devices. The bootloader discovers zpools. For example, "zfs:zroot/kboot-example/boot/kernel" specifies a kernel to boot. Finally, it can be convenient to put the kernel and/or bootloader directly in the Linux initrd. The bootloader itself uses this feature to get necessary data from the /sys and /proc filesystems. Any mounted filesystem can be accessed with "host:<path-to-file>". So, you could boot from "host:/freebsd/boot/kernel" or read the Linux memory usage with "more host:/proc/iomem." The loader also supports mapping the "/sys/" or "/proc/" prefixes to the host's /sys and /proc filesystems, regardless of active device.

Loader.kboot can replace /sbin/init inside the Linux initrd. Init is the first program to run and must do extra steps to prepare the system. Loader.kboot will notice when it is running as init and do these extra steps before starting: mounting all the initial filesystems (/dev, /sys, /proc, /tmp, /var), creating a number of expected symbolic links, and opening stdin, stdout and stderr. The loader runs either in this environment or as a process launched from one of the standard Linux startup scripts. At present, loader.kboot is unable to fork and execute Linux commands.

Refactoring UEFI Booting

Reflecting the long history of booting in general, FreeBSD's boot process had co-evolved with its kernel for the past 30 years in the case of amd64 and for the past 20 years or so in the case of aarch64. Neither of these architectures has been around for this entire time, of course, but amd64 inherited many of the idiosyncrasies of i386, and aarch64's boot, while vastly cleaner, is the product of 20 years of embedded FreeBSD systems. To boot successfully in this complex environment, loader.kboot needed to recreate all these quirks. It follows the UEFI protocols by creating the same metadata structures that loader.efi, our UEFI bootloader, creates.

These efforts started with amd64 since it was more readily available for experimentation. I selected the UEFI + ACPI boot environment to emulate. UEFI was the newer, more flexible interface and seemed to have fewer special cases. While theoretically the FreeBSD kernel could boot from either UEFI or CSM and not know which one it booted from, the practical reality differed. The kernel expected to get UEFI-derived data in certain ways, and BIOS-derived data in slightly different ways. It became clear early on that trying to support both was limiting progress as oftentimes two different paths needed to be written and debugged. Since UEFI will be with us for a long time (even though with LinuxBoot only a tiny sliver of UEFI survives), and CSM might not be, I decided to limit my support to UEFI on amd64.

The efforts started with amd64 since it was more readily available for experimentation.

Despite this simplification, progress was still too slow as I had to discover via trial and error all the quirks the amd64 kernel depended on from the bootloader. Fellow FreeBSD developer Mark Johnston suggested that aarch64 would be easier to get working, since that had a simpler set of interfaces. This proved to be true. Once I had the basic translation from the UEFI data structures to FreeBSD's loader metadata working, I got much further booting aarch64. There were only a couple of bugs that I will talk about later. I'm planning on getting amd64 working next now that aarch64 works.

The loader needs a few hundred lines of code to set up the metadata from UEFI. This code, unsurprisingly, expects to run in a UEFI runtime. It allocates memory using UEFI APIs, gets memory information from UEFI, and fetches the ACPI tables in ways specific to UEFI. I needed to refactor this code so that it could create the proper metadata structures from the /sys and /proc filesystems on Linux. In addition, Linux provides both FDT and ACPI data with device descriptions in ACPI only. This tricked FreeBSD into thinking no devices are present, however, since it favored FDT for device enumeration when both are present. Linux only provides data necessary to do another kexec via FDT, but no device data.

In addition to the normal UEFI data structures and booting, after the kexec, Linux leaves the hardware in a state subtly different from either a cold or warm boot from the firmware. So the system is not quite in a fully reset state. Normally, this doesn't matter—we can boot from that state and put all hardware into its correct state. But there were problems.

UEFI Boot Services was my first problem. When Linux exits UEFI's "boot services," it creates memory mappings where the virtual address (VA) does not match the physical address (PA). FreeBSD's loader.efi always creates 1:1 mappings where PA is equal to VA (so called PA = VA). Since the memory mapping may only be set once, FreeBSD's kernel must use the map Linux creates. The kernel would panic because the mapping was not PA = VA. Fortunately, the panics were due to restrictive assertions left over from debugging loader.efi. Removing the assertions exposed a bug where PA was used instead of VA, but the kernel booted after I fixed that bug.

The second problem I encountered was the "gicv3" issue, where "almost reset" coupled with a device erratum spelled big trouble. There is a design flaw with the gicv3 interrupt router: once it has been started (and Linux starts it fully when booting), it can't be stopped short of a full system reset and initialization (which kexec cannot do). To work around this problem, the FreeBSD kernel had to reuse this memory. Linux passes the gicv3 state data via the UEFI system table structure. This table contains a list of physical addresses reserved for gicv3 use. FreeBSD parses this table, ensures it matches what the gicv3 is using, and marks them as reserved so FreeBSD's memory allocation code doesn't hand them out. Everything worked with QEMU; however, when we tried to run in an aarch64 machine, I was very surprised to discover this problem. Thankfully, the Linux community had previously discovered this problem and had a set of patches that I could use to fix FreeBSD in a similar way.

One not so surprising thing that I discovered during this project was that the bootloader had a lot of copied and pasted code to implement path and device name parsing.

Retiring Technical Debt

One not so surprising thing that I discovered during this project was that the bootloader had a lot of copied and pasted code to implement path and device name parsing. Copies of this code differed in non-obvious ways. Sometimes the changes were bug fixes, but software archaeology showed other copies retained the bugs. Other times, new bugs were introduced in the copies. It's understandable that this would be the fate of the loader. When porting to a new platform, it's easy just to copy code from a working loader and tweak it a little for the new environment. Little thought was given to the long-term effects of lack of refactoring. Once the loader could boot a kernel, why spend more time on the loader? Turns out that this strategy and these attitudes were harmful. The filename parsing code, for example, had been copied from environment to environment so that there were about 10 copies when I started. A common routine was needed for them all—well, all but one, which had a legitimate reason for differing since its device specification varied from the usual "diskXpY:" used elsewhere. Bugs in parsing led me to refactor all this code into one location (so I could fix bugs once). This allowed the novel use of "/dev/XXXX" to work for the "device name" when accessing raw devices. It also lets "host" prefix work without needing a unit number. The loader's filename parser is way more flexible than I'd ever known.

Conclusion

Adapting the FreeBSD bootloader and kernel to boot in this new environment proved to be straightforward. The large number of small tasks for Linux host integration, coupled with FreeBSD's undocumented bootloader-to-kernel handoff, provided the biggest challenges for this phase of the project. The unexpected hardware erratum added to the excitement and caused the largest change needed to the kernel. With FreeBSD/aarch64 successfully booting on real hardware, we can proceed to the next phase of the project: creating our own firmware and finding the remaining FreeBSD/amd64 bugs. Even with these limitations, we have used loader.kboot to download an installer ramdisk to provision a system and reboot the result. It has also been incorporated into last summer's loader continuous integration GSoC project.

Next Up

In the next article, we'll create a LinuxBoot firmware image that boots FreeBSD. I'll explain how firmware images are packaged, the tools used to create and manipulate them, and how to reflash your firmware images if you are brave enough. I'll help you select the right tools from the many options Linux provides for creating an initrd, and give a sample script to find and boot your FreeBSD system. With luck, you'll have a simpler, faster, more secure firmware by the end.

WARNER LOSH has been contributing to the FreeBSD project for many years. He's contributed many features and fixes to the bootloader. His interest in Unix history extends to how bootstrapping evolved alongside Unix and its many derivatives. He lives in Colorado with his wife Lindy, who loves drawing cats and dogs, and his daughter, who plays an assortment of brass instruments. He can often be found walking dachshunds.