Kboot: Booting FreeBSD With LinuxBoot

EuroBSDCon 2023
Coimbra Portugal
September 16, 2023

Warner Losh, Netflix

Outline

- History of Booting With Linux
- LinuxBoot Project
- Early FreeBSD efforts: kboot and ps/3 port
- LinuxBoot Today
- FreeBSD’s UEFI booting overview
- LinuxBoot Environment
- Adopting kboot to uefi + amd64 and aarch64
- FreeBSD kernel changes
- Demo
- The Big Logo Reveal

Prehistory: LinuxBIOS and coreboot

LinuxBIOS
(1999)

coreboot
(2008)

LinuxBIOS: Ron Minnich

Los Alamos 1990s

Evolved into coreboot

https://coreboot.org/

Chromebooks and open laptops

Source: https://trmm.net/LinuxBoot_34c3/

Introduction to Booting with Linux 2004-2015

- kexec_load(2) system call (since Linux 2.6.6) 2004
  - Though out of tree patches were common for sometime before that
- Originally just for chain booting next kernel faster
  - Faster boot, less downtime
- Added crash kernel support (2.6.13) 2005
  - Reboot to crash kernel which then saved the crashed kernel’s state
- Added hibernation support (2.6.27) 2006
  - For suspend / resume support: save current in-use memory, read it back later
- Later exec_load_file(2) for secure booting (2008)
- kexec-tools

Typical Uses of Kexec booting (2004-2015)

- Used to make reboot faster on x86 (more 9’s in uptime)
- Used for crash kernel
- Used for more reliable embedded rebooting
- Used by some hypervisors to constrain what could be loaded
- Used kexec-tools by and large
- Other tools slowly developed
  - Early stages of uroot and similar environments date to this time

Linux on PS/3 (2011-2015)

- Could only boot via kexec
- Many different programs around to do this
- Kboot was one created for Ubuntu on PS/3 project
- Petiboot was another (that would go on to have life in later PowerPC hypervisors in Power8 and Power9)
  - https://github.com/open-power/petitboot

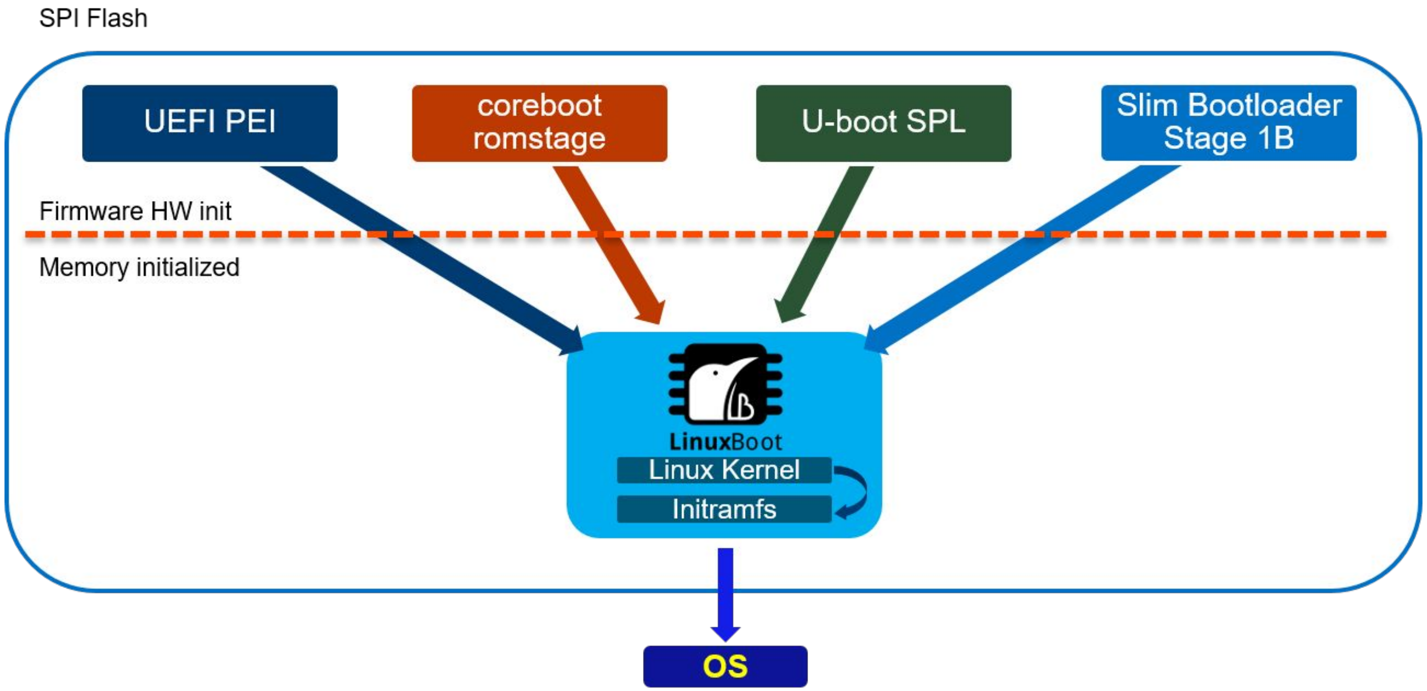
LinuxBoot: Because All That Stuff Is Too Complicated

- Better pre-boot Environment than UEFI for complicated scripting
  - Boot No Matter What (to report to control plane problems)
  - Fetching a signed image if the OS is damaged beyond booting
  - Making other boot-time decisions
  - Reprovisioning
  - Decommissioning
  - Booting the Normal OS
  - Running Advanced Diagnostics Inappropriate for Full OS
  - Cope with missing / failing persistent media (eg NVMe, HDD, etc)
- Gaining Popularity in Embedded Space
- Can also be useful to deploy FreeBSD in Linux Only VM setups
  - Assuming the provider's kernel has the right config options enabled

The LinuxBoot Project (2017 - Present)

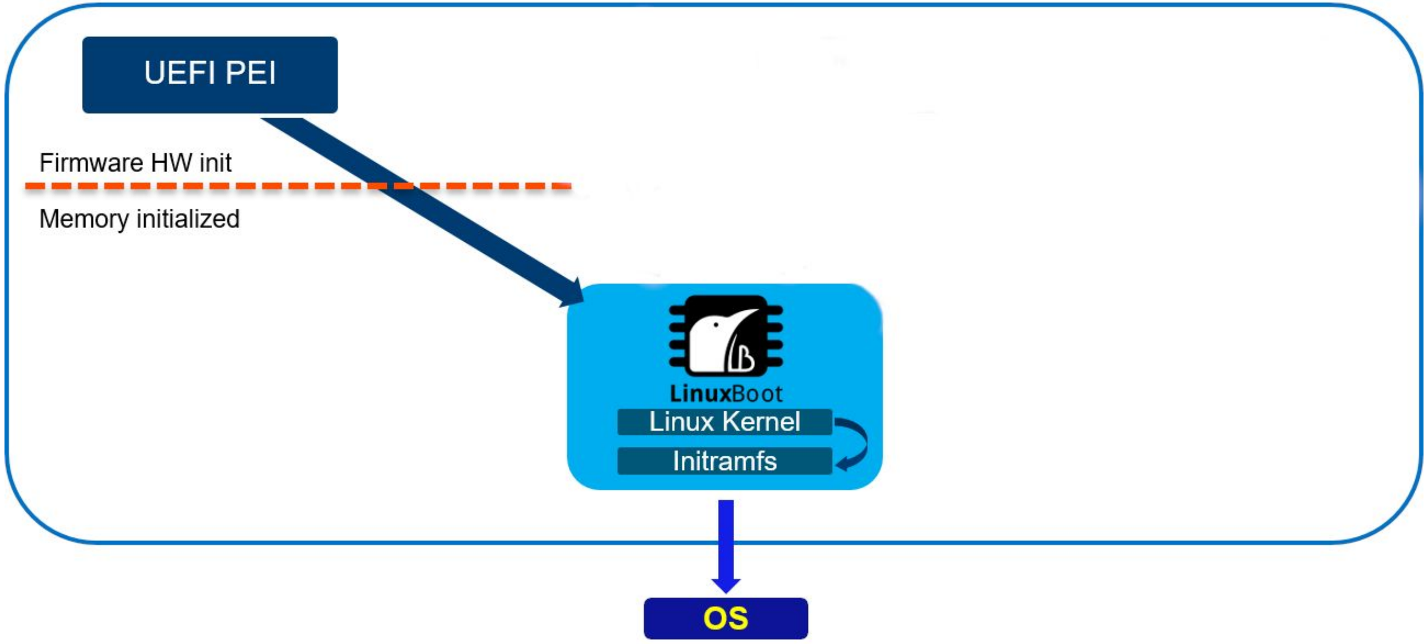
- Started at Google as NERF
- Created to provide a unified booting environment
- Also created to get rid of UEFI (though not 100%)
- Theory of Operation
  - Low level boot loader will initialize a machine just enough for Linux
  - Linux will take over and run scripts to figure out what to boot
    - And then will use kexec to boot that
- Brought together UEFI, U-BOOT, coreboot and Slim Bootloader
- Very Helpful Community… Worth the effort to find them…

LinuxBoot Block Diagram



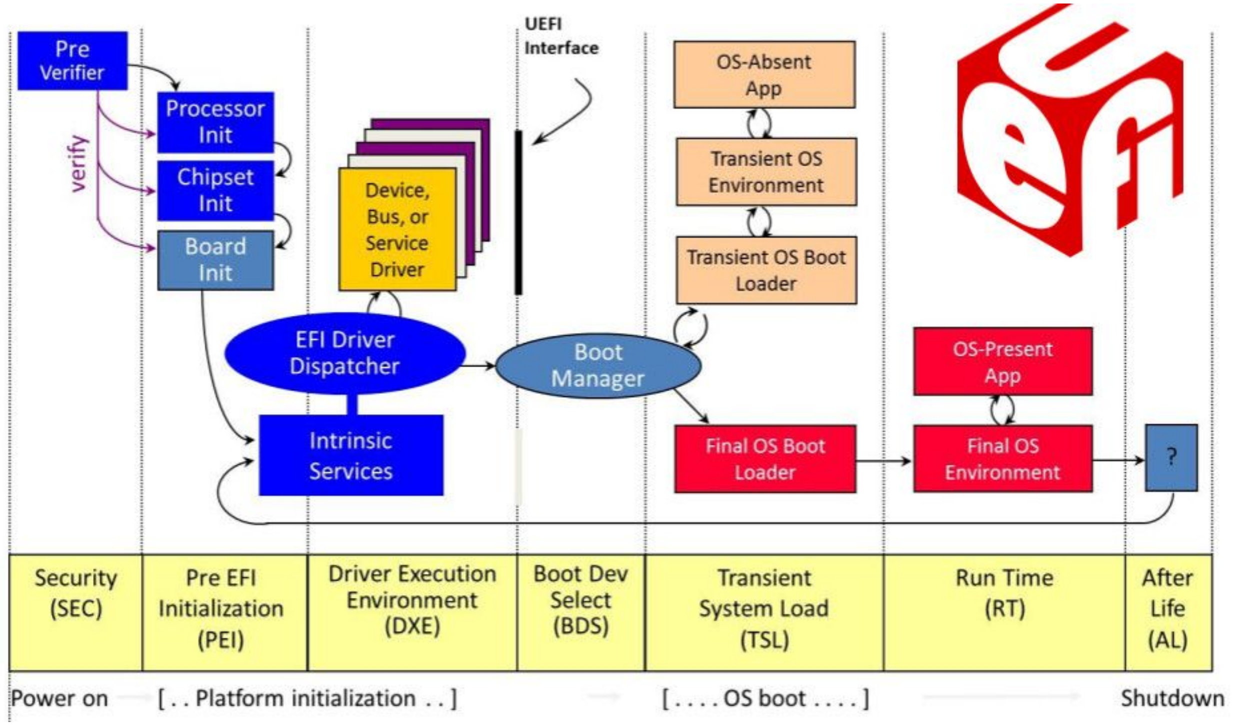
Source: linuxboot.org

LinuxBoot Block Diagram (for this talk)



Source: linuxboot.org

Starting Point for LinuxBoot

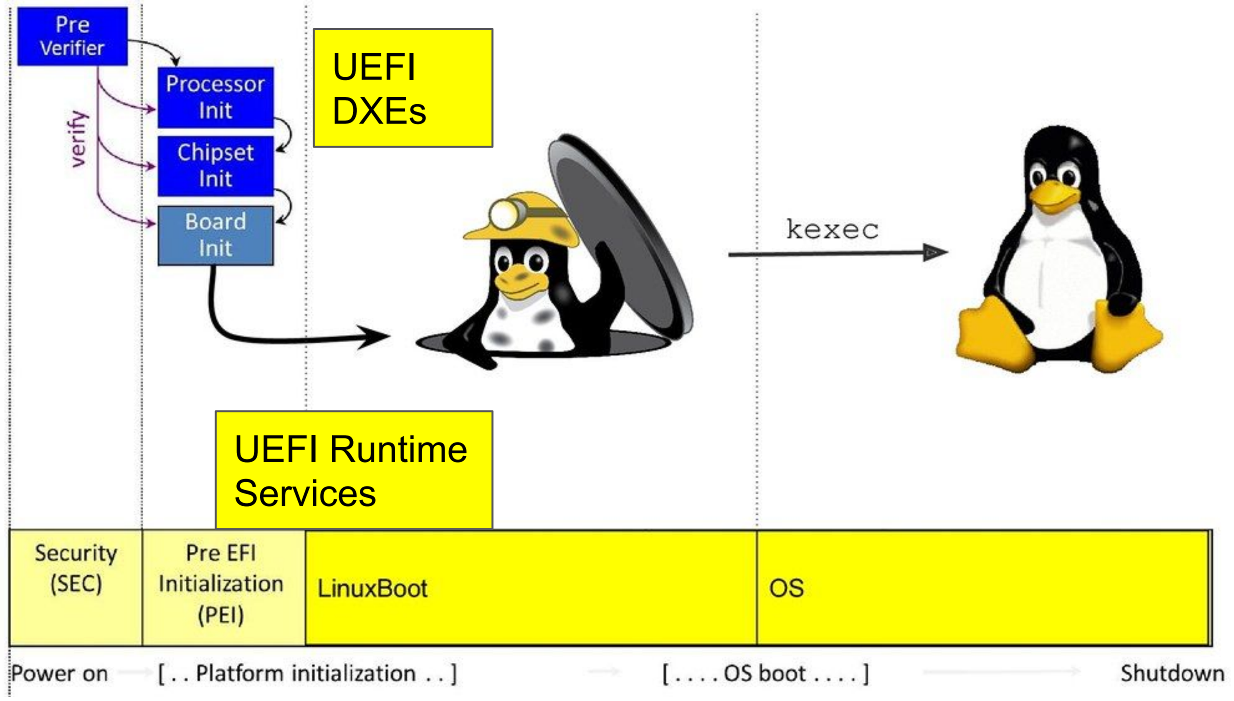


Source: https://trmm.net/LinuxBoot_34c3/

“Mainline” Booting With UEFI

- Processor Establishes Root of Trust
- Processor’s early boot code runs just enough to load UEFI’s PEI
    - PEI - Pre EFI Interface
- The PEI code initializes low level details of the machine
    - Memory
    - Other devices / clocks / etc
- Hands off control to UEFI DXE (Device Execution Environment)
    - Discovers bootable devices
    - Initializes all UEFI drivers, network stacks, storage stacks, etc
- Starts BDS
    - Selects which device to load from
    - Loads some bootloader.efi
        - loads the kernel

LinuxBoot Final State



Source: https://trmm.net/LinuxBoot_34c3/

Security Hardening

- LinuxBoot removes much of UEFI
  - In EDK2, only the Pre EFI Initialization (PEI) is left and RuntimeServices
  - Almost all device drivers eliminated
  - Network and kernel stack eliminated
- Uses boot device selection (BDS) for x86_64/amd64
  - Linux kernel + UEFI capsule
- Uses shell.efi for all other platforms
- Reproducible builds

Advantages to a LinuxBoot

- Security Hardened Setup
  - Only uses minimal amounts of UEFI firmware
  - Moves the rest to a Linux kernel
- Only write device drivers once – Linux yes, UEFI DXE no
  - Very important for new SoCs: faster time to market
  - Less important for x86 or older SoCs (since drivers already written)
- Some environments LinuxBoot Only now (that is, you have no choice)
- Better pre-boot environment than with UEFI EDK2
- Linux is more security hardened than UEFI
  - Linux has 100k's of contributors
  - Grub / EDK2 each have 100-200
- Often will boot faster than EDK + boot loader + kernel (demo later)

LinuxBoot tl;dr

- Combines UEFI PEI code with Linux Kernel with UEFI capsule
- Normal Linux kernel + initramfs
- Linux rc scripts finds real kernel and kexec(2)s it
- More Secure, Better Audited, Less Attack Surface

FreeBSD initial kboot: The PS/3 Port

- Nathan Whitehorn write initial kboot for PS/3 port 2015
  - Fairly simple, just loaded the kernel
  - Used DTB to find memory
  - Was specific to PS/3, petiboot, etc
- Expanded for other PowerPC, especially Power7 and Power8
- Very little metadata was passed in
- Limited support for loading modules, etc
- FreeBSD/powerpc boot handoff interface simplified for direct petiboot handoff
  - No modules could be loaded or tunables set
  - PowerPC could get all it needed w/o the need for a loader
- After the world moved to Petiboot, FreeBSD's kboot languished

Why not direct boot like PowerPC

- PowerPC, OpenFirmware can get all the data the kernel needs
- AMD64 and aarch64 we have a co-evolved interface
  - Tunables
  - Loadable modules
  - Memory maps
  - ACPI tables
  - EFI handle
  - SMBIOS
- Can’t boot the amd64 kernel directly ever (panic with no SMAP)
- Aarch64 has limited support for direct boot, but needs U-BOOT’s DTB pointer for memory map, etc

FreeBSD UEFI Booting Sequence (amd64)

- Boot1.efi -> Loader.efi -> kernel
  -> drivers
  -> metadata
  -> memory tables
- Kernel used to start in 32-bit mode so the pointers are 32-bit
- Can load anywhere, page tables to pass in the PA of the kernel
- Kernel can’t call BIOS, so loader must do so and pass needed data to kernel
- Boot Loader provides same metadata for both BIOS and UEFI [*]

FreeBSD UEFI Booting Sequence (aarch64)

- Simpler handoff
- Turns off MMU to find its PA (and to work w/o MMU)
- Loader.efi -> kernel
  - -> drivers
  - -> metadata
  - -> memory tables
- Passes in memory, UEFI tables, ACPI, etc
- Fewer required data setup, but some data from before ExitBootServices

LinuxBoot startup

- Mounts the initramfs image as /
- Runs init
- Runs /etc/rc
- System setup to read the new kernel
- Some variation on kexec
  - For me it’s loader.kboot

loader.kboot

- FreeBSD’s /boot/loader compiled as a linux binary
  - We have a mini linux libc and linker scripts
- “Stand” devices are written to access host (linuxboot) resources
  - Host disk access (to find FreeBSD images)
  - Host filesystem access (to get the data the boot loader needs)
    - Normal “open” just works
- Otherwise, it’s just the bootloader you are familiar with…

Changes needed

- Refactor the device naming support
  - Since I wanted to really use natural names for host and target files at the same time
    - host:/proc/iomem
    - /dev/nvme0ns1:/boot/kernel/kernel
  - An area that suffered way too much cut and paste
  - Unified code shows we can use increasingly flexible names.
- Host Block support (read data from /sys/block to populate)
  - Original code didn’t have lsdev support
- Refactor loader UEFI support
  - Reuse metadata provisioning from UEFI
  - Many years of copying of code without thinking about refactoring

What is technical debt?

Source: https://melv1n.com/what-is-technical-debt/

Loader.kboot Information Gathering

- Get Memory Layout
  - Look in host:/sys/firmware/dtb (aarch64) to find PA of table
    - Open /dev/mem and read them out to attach to metadata
    - Or pass different metadata to kernel and have trampoline fill them in
  - Look in host:/sys/firmware/memmap (amd64)
    - More direct, supposed to be MI, but isn’t
  - Both use host:/proc/iomem as unreliable backup (or could)
- Get ACPI table base
  - Look in host:/sys/firmware/efi/systab (same as for SMBIOS addr).
  - or look in host:/sys/firmware/smbios
- EFI Video modes
  - Aren’t shared / supported now
- DTB
  - Passed in, but need to favor ACPI devices over DTB
    - Should check for simpelbus in dtb before preferring it over ACPI

UEFI Memory Maps

- Linux EFI Stub calls ExitBootServices
- Linux calls runtime_services->SetVirtualAddressMapping
  - Routine can be called only once.
- Linux exports UEFI Memory Table in different ways
- loader.kboot must pass this to the kernel
- FreeBSD can no longer assume UEFI calls are made VA == PA
  - Mostly involves removing asserts and fixing comments
  - FreeBSD's UEFI loader.efi calls SetVirtualAddressMapping with VA == PA
- Becomes very important on aarch64 for GICv3 workaround

Kexec_load(2) Interface

- `long kexec_load(unsigned long entry, unsigned long nr_segments, struct kexec_segment *segments, unsigned long flags);`
  - Available if CONFIG_KEXEC=y in kernel config
  - Up to 16 segments can be loaded
  - Flags control
    - CRASH kernel or replacement kernel
    - Whether or not to reset hardware context
- Once loaded, the image waits for
  - A crash (if a crash kernel)
  - reboot(2) with special magic flags (CMD_KEXEC) to restart it instead of rebooting to bootloader
- Entry is a pointer to a routine (presumably) in the image that handles booting the kernel (more later)
- Note: there's other details and complications, but it's not relevant to this project

Starting the New Kernel

1. Find a ‘big enough’ area of memory to load everything
    a. Limited number of slots, and linux mmaps high -> low so have to go big or lots of copying
2. Load the kernel into malloc’d area
3. Add the metadata, tuneables, memory tables, etc
4. Add the loadable modules
5. Make page tables for PA to VA (KERNBASE) mapping
6. Create a mapping table from the VA in loader.kboot for each of the segments
    a. one for the trampoline page(s)
        i. All data and code to start the kernel (incl page tables)
    b. one or more for the kernel and modules
        i. This whole area must be contiguous in PA
7. Load it with kexec_load(8)
8. Call reboot(CMD_KEXEC)

Trampoline to Kernel

1. amd64
   a. Load CR3
   b. Turn on address translation
   c. Jump to btext (start field in elf binary)
2. aarch64
   a. Load page table
   b. Copy UEFI memory tables (if needed)
   c. Jump to _start
3. Kernel takes over just as if it had been booted directly with UEFI

```
        .text
        .globl  tramp
tramp:
        cli                             /* Make sure we don't get interrupted. */
        leaq    tramp_pt4(%rip), %rsp   /* Setup our pre-filled-in stack */
        popq    %rax                    /* Pop off the PT4 ptr for %cr3 */
        movq    %rax, %cr3              /* set the page table */
        retq                            /* Return addr and args already on stack */
/*
 * The following is the stack for the above code. The stack will increase in
 * address as things are popped off of it, so we start with the stack pointing
 * to tramp_pt4.
 */
        .p2align        3               /* Stack has to be 8 byte aligned */
trampoline_data:
tramp_pt4:      .quad   0               /* New %cr3 value */
tramp_entry:    .quad   0               /* Entry to kernel (btext) */
        /* %rsp points here on entry to amd64 kernel's btext */
                .long   0               /* 0 filler, ignored (current loaders set to 0) */
tramp_modulep:  .long   0               /* 4 moudlep */
tramp_kernend:  .long   0               /* 8 kernend */
                .long   0               /* 12 alignment filler (also 0) */
tramp_end:
        .text
        .globl  tramp
tramp:
        adr     x8, trampoline_data
        ldr     x10, [x8, #TRAMP_MEMMAP_SRC]
        cmp     x10, xzr
        b.eq    9f

        /*
         * Copy over the memory map into area we have reserved for it. Assume
         * the copy is a multiple of 8, since we know table entries are made up
         * of several 64-bit quantities.
         */
        ldp     x11, x12, [x8, #TRAMP_MEMMAP_DST]       /* x12 = len */
1:
        ldr     x13, [x10], #8
        str     x13, [x11], #8
        subs    x12, x12, #8
        b.hi    1b
9:
        ldp     x9, x0, [x8, #TRAMP_ENTRY]              /* x0 = modulep */
        br      x9
```

Aarch64 GICv3 issue

- GICv3 has design flaw (can’t be stopped and restarted after warm reset)
- Have to use same memory that Linux used after it initialized it
- Linux passes these addresses in via UEFI reserved tables
- FreeBSD had to notice GIC has started and initialize differently
  - Use the pending and other tables already programmed in
  - Make sure that memory is marked reserved in table
  - Linux passes this in via UEFI system tables that are Linux specific
- Once running, things are the same
- Code written, but not yet upstreamed
  - Need to address some review feedback and I’m unsure the best way

FreeBSD kernel changes

- Had to change UEFI to not assume PA = VA
  - The Linux UEFI stub exit boot services and sets up different mapping and then calls SetVirtualAddressMap
    - This can be called only once per boot
    - Can only be called after you exit boot services
    - We have to eat what we're given
    - Loader.efi calls it, but with tables where PA = VA
  - When creating the PMAP for calls to UEFI, I had to remove PA = VA asserts now not an error
- Had to fix the GICv3 interrupt controller
  - Due to a design defect, it's impossible to restart
    - Have to use the memory already allocated for it
    - Must not use that memory
  - Fixed places in the gicv3 code that assumed warm reset state for device
- Fix issues with FDT vs ACPI device tree selection
  - Could be a lot better (I fixed the selector, but smarter selection needed)

Running as init

- If we're pid 1
  - Create directories for mount points
  - Mount special filesystems: /sys, /proc, /dev, /tmp
  - Make symlinks from /dev/std{in,out,err} to /dev/fd/{0,1,2}
  - /var/tmp, /var/lock
  - Symlink /run to /tmp
  - Setup tty so /dev/console is FD 0, 1, 2
- Future Needs
  - Initialize Signals
  - Cope with having children
  - Needed so fork/exec to support io.popen() or os.system() work
  - SIGSEGV/SIGBUS need handlers for traceback / crash info

TODO: What's left

- Upstream the gicv3 and PA != VA patches
- Fix getting time via UEFI
- Fix a half dozen other annoying, small but important bugs
- Finish amd64
- Refactor the handoff
  - Code sharing by copying and ifdef should move to code sharing
- Increase lua bindings for host things
  - io. and os. libraries, for example
- Fix the console problem (though only slightly related to kboot)

ZFSBootMenu

https://github.com/zbm-dev/zfsbootmenu

FreeBSD boot loader inspired port to Linux

Will add support to using loader.kboot to allow FreeBSD booting

Very easy people to work with

Detour: Booting Windows from LinuxBoot

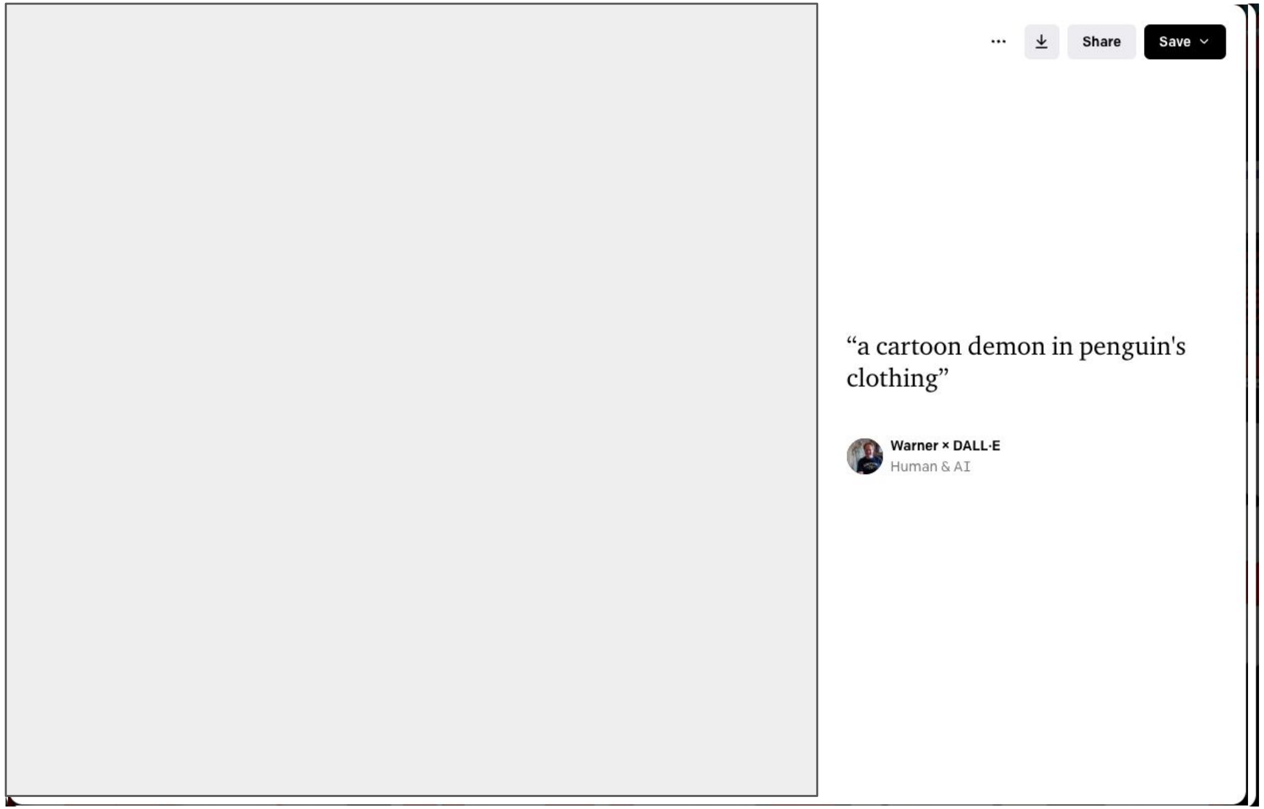
- Can’t just ‘chain boot with GRUB’ since EFI has exited boot services (Windows Requires it)
- Windows via POC: change kexec to run .efi PEs. Runs bootmgfw.efi which runs winload.efi which boots Windows… Ofir Weisse summer intern at Google.
- Changes kexec to load and run UEFI binary
- https://www.osfc.io/2019/talks/booting-windows-on-linuxboot/
- Now abandoned….
- Newer work https://www.osfc.io/2022/talks/linux-as-a-uefi-bootloader-and-kexecing-windows/
- Runs Linux in an environment that uses UEFI drivers, doesn’t exit boot services and has other restrictions to make it possible to exit out of the linux kernel to boot Windows later.

Final Thoughts

- FreeBSD is the only non-linux, non-grub-assisted fully booting UEFI OS under LinuxBoot on x86 or aarch64
  - Though some things like plan9 have booted as science experiments.
  - UEFI, especially on aarch64 and amd64, require special handoffs
  - X86 BIOS booting of really old FreeBSD or NetBSD kernels may also be an option
- Potential uses for this are diverse

And Now The FUN Stuff: The LOGO

Most Important Thing: A Mascot



“a cartoon demon in penguin's clothing”

Warner × DALL·E
Human & AI

The Winner

Questions

Warner Losh

imp@freebsd.org

wlosh@netflix.com