

ECE 598 – Advanced Operating Systems Lecture 3

Vince Weaver

`http://web.eece.maine.edu/~vweaver`

`vincent.weaver@maine.edu`

30 January 2018

Announcements

- HW#1 was posted, due on Thursday.
- Next homework will use the Pi
- Getting things going on Pi3 a lot harder than expected.



Booting a System

- Why is it called booting?
- Most likely source is the idea of “Pulling oneself up by ones bootstraps”, i.e., getting somewhere by starting with nothing



Simple Booting

- Simplest systems have code in ROM.
The CPU initializes, points the Program Counter to a known location, and starts executing.
- The STM32L boards in ECE271 do something similar; code is in flash, reset vector (at offset 0) points at code to start. press reset, runs reset vector, up to you to do everything else.

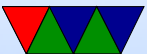


Firmware

- Low-level code (often written in assembly language) that initializes the system.
- Often in ROM/EEPROM/FLASH
- Boot firmware initializes system.
 - Init RAM? Set it up (often over i2c), clear out random or old contents (if a soft reboot). This part operates without memory or stack to use, tricky.
 - Init other hardware. I/O, serial ports, keyboard, display, etc.



- Load code to boot. Where from? Hard-disk, floppy, network (PXE), CD/DVD, USB, SD-card, etc. Old days: tape, paper-tape, console switches?
- Might have other interfaces: boot selection/configuration screen?
- Some firmware provides routines for hardware to use, for things like accessing disks, writing to screen, reading keyboard, initializing security, etc.
- Firmware development is hard. Not all corner-cases well tested (can it boot Windows? Ship it). Kernel and Firmware devels have antagonistic relationship.



Booting on x86

- Firmware
 - BIOS original firmware. 16-bit. Dates back to CP/M days. Provided booting and a library for accessing I/O. (MS-DOS a thin layer over BIOS).
 - These days EFI and uEFI replacing it, 32/64-bit. Written in higher level language.
 - Firmware provides other interfaces, like power management, ACPI, device enumeration, etc.
 - x86 firmware can use SMM mode which allows



secret/hidden code running behind the scenes for things like hardware emulation (USB keyboards) and power management.

- Firmware traditionally loaded a 512Byte bootsector (last two bytes 0x55 0xAA) to 0x7c00 and jumped to it. This “first stage” then had enough code to load a more complex “second stage”
- The bootloader (GRUB is common on x86 Linux) then loads the operating system. Provides nice graphical interface often (to select images) and a console for command line arguments and browsing kernel images.



Loading Linux

- Linux is usually on disk, sometimes a separate boot partition. Complicated because blocks might not be contiguous on disk.
- Some Linux images can be loaded directly, without need of bootloader.
- Linux image itself can be complex
 - Linux image, “vmlinux” (why called that? historical, unix, vm unix)
 - decompressor and compressed image (zImage,



bzImage, uImage, etc)

- When building, the kernel image is taken, stripped, compressed. piggy “piggyback” code put on, as well as decompressor. Originally floppy boot code stuck on beginning as well.
- Different entry points. On x86 BIOS boots into 16-bit. EFI and bootloaders can jump into 32/64
- So optionally boots in 16-bit mode. Switches to 32-bit mode. If 64-bit, optionally switch to 64-bit Decompressed kernel to 0x10 0000 (might have to move decompress code). (above 1MB. Why? 640k)



What about initrd?

- Jump to startup_32 / startup_64 function
- 16-bit code handles various stuff, gets memory size from BIOS, etc
- 32/64 relies more on boot loader. Has specification for how registers set up, etc.
- relocates decompression code if needed. Sets up stack, clears BSS, Decompresses.
- relocate if needed. why? randomization is one.
- Memory map. Virtual mem. First 896M of physical mem mirrored in top of 32-bit. Why? So kernel can



easily copy to/from. Can convert kernel virt to phys with just subtraction. Complicated if more than that much RAM, have to copy around. HIGHMEM.

- space above for vmalloc
- somewhat more complicated 64-bit
- kernel just an ELF executable
- Starts Userspace program “init” (old days simple program and shell scripts, these days “systemd”)
- Sometimes an “initrd” is included too that has enough drivers to get Linux going and a very minimal filesystem to help with booting before disks/filesystem ready.



Disk Partitions

- Master Boot Record, Boot Sector
- Followed by partition table
- Way to virtually split up disk.
- DOS GPT – old partition type, in MBR. Start/stop sectors, type
- Types: Linux, swap, DOS, etc
- GPT had 4 primary and then more secondary
- Lots of different schemes (each OS has own, Linux supports many). UEFI more flexible, greater than 2TB



Bootloaders on ARM

- The most common is uBoot
- uBoot – Universal Bootloader, for ARM and other embedded systems
- Almost like minimal OS
- More of a challenge to write a bootloader for a widely nonstandardized architecture like ARM. (Why is ARM so nonstandardized?)



Uboot Booting

- Most other ARM devices, ARM chip runs first-stage boot loader (often MLO) and second-stage (uboot)
- FAT partition
Why FAT? (Simple, Low-memory, Works on most machines, In theory no patents despite MS's best attempts (see exfat))
The boot firmware (burned into the CPU) is smart enough to mount a FAT partition



Booting on typical ARM/u-boot

- vmlinux. strip. compress. piggy / piggyback. convert to zImage. mkuimage converts to uimage suitable for booting with u-boot
- No bios really. Bootloader provides all info.
- Device Tree provides config info for hardware (memory size, interrupts, what hardware is there). This allows kernel that will run on many ARM boards (PI, beaglebone, pandaboard, etc) rather than having to have a different hard-coded kernel for each possible platform.



Kernel booting

- Initializes hardware. First part asm. Transition to C as quickly as possible. First thing to initialize. Memory. Then simple in/out. Enable keyboard, simple VGA, serial console. So printk can work.
- Relocates decompression code
- Decompresses
- Parse the resulting ELF file.
- Apply any relocations
- Jump to entry point



Raspberry Pi Booting

- Unusual (and has changed over the past few years)
- Small amount of firmware on SoC
- Pi is actually a large GPU chip with helper ARM chips
 - VPU – Dual Core Videocore IV chip CPU, SIMD, Parallel Processor, ThreadX OS, co-ordinates everything, as well as video codecs, power, etc.
 - ISP – Image Sensor pipeline, processing for the cameras
 - QPU – Quad Processor Unit – 24 GFLOP compute pipeline, co-ordinate and vertex shader.



Raspberry Pi Booting, More Details

- Power on – First stage – Boot ROM in GPU starts up GPU runs things. Explains rainbow pattern, lightning bolts, etc
- First stage tries to load `bootcode.bin` second stage
 - On original Pi this was just SD card
 - Pi3 can boot off of Secondary SD, SPI, NAND, USB or network.

Also some complex hack to flash the OTP (?) to allow GPIOs to be dedicated to boot-selection



- bootcode.bin loaded into L2 cache (shared CPU/GPU?) and executed
- Second stage – bootcode.bin – binary blob that is loaded from the SD card and run by GPU. 1 million+ lines of code? Mostly written and maintained by one guy at Broadcom?
 - Has own non-ARM assembly language
 - Efforts to reverse engineer:
 - <https://github.com/christinaa/rpi-open-firmware>
 - <https://github.com/hermanhermitage/videocoreiv>
 - Inits SDRAM, gets ARM chips ready (if multiple, puts



- them in low-power sleep loop)
- Third stage – start.elf (cd, db, x)
(used to be an additional stage before this)
Loads and parses config.txt
Lots of settings in config.txt
 - cd = cut down, if only 16MB of GPU memory specified
ARM and GPU share RAM. Cuts out OpenGL, etc.
 - x = extra (have things like video codecs, camera)
 - db = debug, extra asserts
 - Also fixup.dat (cd, db, x) – used for configuring memory split?



- Finally loads kernel. kernel.img (old 1176 Pis), kernel7.img (cortex a7 pis and backwards compat). kernel8.img hack, for 64-bit pi3
cmdline.txt has command line passed to kernel
- Device tree files, .dtb.
When kernel booted r0=?, r1=?, r2=pointer to config
Old fashioned, ATAGS, set of value/data pairs for Linux booting. New: flattened device tree. Standard for all Linux builds.
Other? ACPI is x86 way, new ARM servers support this so Windows can boot.



These are important, have things like location of hardware in memory, which IRQ to use, which GPIOs to use, how much memory is free and where it is, what type of CPU, etc. By standardized format, can have “generic” linux kernel that can run on any ARM generation without having to hard code that all.



So how do we start with own OS?

- Make simple binary.
- Compile it with ARM toolchain (cross compile?)
- Replace kernel.img on your memory card.
- Boot into it!
- Easier said than done.
- What kind of setup do you have?

