

# **ECE 471 – Embedded Systems**

## **Lecture 16**

Vince Weaver

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

`vincent.weaver@maine.edu`

15 October 2018

# Announcements

- Midterm is graded
- Don't forget HW#5
- No class Wednesday, Career Fair



# Go over midterm



# HW#4 Coding Review?

- Error checking! If error, exit. Don't write invalid fd!  
Linux doesn't care, but still not best thing to do.
- Unexport display, what if don't.
- debouncing— both hit and miss
- be sure to close your fd or you'll run out
- running "system()"
- indentation, comments
- Need leading / in front of /sys/
- If you try to compare against uninitialized memory, what



value will it have?

- Doing a `write(fd,"0",0)`
- `linefeed` at end of messages



# Raspberry Pi Booting

- Unusual
- Small amount of firmware on SoC
- ARM 1176 brought up inactive (in reset)
- Videocore loads first stage from ROM
- This reads `bootcode.bin` from fat partition on SD card into L2 cache. It's actually a RTOS (real time OS in own right "ThreadX") (50k)
- This runs on videocard, enables SDRAM, then loads `start.elf` (3M)



- This initializes things, the loads and boots Linux kernel.img. (also reads some config files there first)  
(4M)



# More 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





# Trusted Firmware

- Are all computers general-purpose? Can you lock them down so only the manufacturer provided code runs?
- Why would you want to do that?
- New for ARMv8: ARM Trusted Firmware (ATF). Two standards, vendors have possibly made a mess of it already.



# Boot Methods

Firmware can be quite complex.

- Floppy
- Hard-drive (PATA/SATA/SCSI/RAID)
- CD/DVD
- USB
- Network (PXE/tftp)



- Flash, SD card
- Tape
- Networked tape
- Paper tape? Front-panel switches?

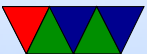


# Disk Partitions

- 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
- Why partition disks?
  - Different filesystems; bootloader can only read FAT?



- Dual/Triple boot (multiple operating systems)
- Old: filesystems can't handle disk size



# Device Detection

- x86, well-known standardized platform. What windows needs to boot. Can auto-discover things like PCI bus, USB. Linux kernel on x86 can boot on most.
- Old ARM, hard-coded. So a rasp-pi kernel only could boot on Rasp-pi. Lots of pound-defined and hard-coded hw info.
- New way, device tree. A blob that describes the hardware. Pass it in with boot loader, and kernel can use



it to determine what hardware is available. So instead of Debian needing to provide 100 kernels, instead just 1 kernel and 100 device tree files that one is chosen at install time.

- Does mean that updating to a new kernel can be a pain.



# Detecting Devices

There are many ways to detect devices

- Guessing/Probing – can be bad if you guess wrong and the hardware reacts poorly to having unexpected data sent to it
- Standards – always knowing that, say, VGA is at address 0xa0000. PCs get by with defacto standards
- Enumerable hardware – busses like USB and PCI allow you to query hardware to find out what it is and where





it is located

- Hard-coding – have a separate kernel for each possible board, with the locations of devices hard-coded in. Not very maintainable in the long run.
- Device Trees – see next slide



# Devicetree

- Traditional Linux ARM support a bit of a copy-paste and `#ifdef` mess
- Each new platform was a compile option. No common code; kernel for pandaboard not run on beagleboard not run on gumstix, etc.
- Work underway to be more like x86 (where until recently due to PC standards a kernel would boot on any x86)
- A “devicetree” passes in enough config info to the kernel



to describe all the hardware available. Thus kernel much more generic

- Still working on issues with this.



# Booting Linux

- Bootloader jumps into OS entry point
- Set Up Virtual Memory
- Setup Interrupts
- Detect Hardware / Install Device Drivers
- Mount filesystems
- Pass control to userspace / call init



- Run init scripts
- rc boot scripts, /etc/rc.local  
Start servers, or “daemons” as they’re called under Linux.
- fork()/exec(), run login, run shell



# How a Program is Loaded on Linux

- Kernel Boots
- `init` started
- `init` calls `fork()`
- child calls `exec()`
- Kernel checks if valid ELF. Passes to loader
- Loader loads it. Clears out BSS. Sets up stack. Jumps



to entry address (specified by executable)

- Program runs until complete.
- Parent process returned to if waiting. Otherwise, init.



# Viewing Processes

- You can use `top` to see what processes are currently running
- Also `ps` but that's a bit harder to use.

