ECE 471 – Embedded Systems
Lecture 16

Vince Weaver

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

vincent.weaver@maine.edu

5 October 2022

Announcements

- Midterm on 14th
- Don't forget HW#5
- No need to hand displays back yet

Homework #4 Error Checking

- What do you do if there's an error?
- Ignore it? Why could that be bad?
- Retry until it succeeds?
- Print an error message and continue?
  Can you continue?
  What if continuing with a bad file descriptor breaks things?
  What if printing too many error messages fills up a log, swamps the screen, hides other errors?

- Good error message
  Can't be confused with valid input (airlock)
  If displayed to user, make it easy to understand
- Print an error message and exit?
  What if it's a critical system?
- Crashing is almost never the right answer.
- Can get more info on error with errno / strerror()

Homework #4 Permissions

- We haven't really discussed Linux permissions
- List file, "user" "group" "all"
- `drwxr-xr-x`
- Often in octal, 777 means everyone access
- Devices under /dev or /sysfs might be set to only root or superuser
- Traditional UNIX /dev you can set with chown (to set user/group) or chmod (to set permissions)
- Group under /etc/group, so gpio group

- Why is it better than using “sudo”? Why might I as grader not want to run your code using “sudo” if I can avoid it?
- How to set up sudo? /etc/sudoers file

Homework #4 – LED Blinking

- Blink frequency. Remember, 1Hz is 500ms on / 500ms off
  not 500us, not 1s
- Blink correct GPIO. Does it matter? Want to fire engines, not engage self destruct.

Homework #4 – Switch

- Debouncing
  - 100ms or even 10ms is long time
  - Tricky as we are detecting levels not edges here
  - Reading and only reporting if you say have 3 in a row of save val
  - Reading, sleeping a bit, then report the value after has settled
  - Just sleeping a long time after any change? If a short glitch happens this might misreport.

- Sleep too long, might miss events
- Debounce if using interrupt-driven code

  In that case debouncing might be to ignore repeated changes if they happen too close together

Homework #4 – Something Cool

- How can you read/write at same time (say to let switch activate LED)
- Need to make copy of data structures
- If you do re-use, make sure you close(), especially if you open multiple times. Either will get EBUSY or else fd leak

Homework #4 Question

- 5.a Why usleep? Less resources (not busy sleeping), cross-platform (not speed-of-machine-dependent), compiler won't remove, other things can run, power saving.
  Be careful saying accuracy! usleep() guarantees a minimum time delay, but it is best effort how long the delay actually is. So if you really need *exact* time delays you probably want some other interface.
- 5.b Layer of abstraction. In this case, not having

to bitbang the interface or know low-level addresses, portability among machines.
ability to run WiringPi is not a benefit

- 6.a Machines from dmesg: 2022: Pi4 (3) Pi3B+ (1)
  dmesg a good place to find error messages, etc. grep
- 6.b Kernel versions. Current Linus kernel (upstream) is 6.0
  Uname syscall, what the parts mean

```
Linux linpack-test 4.14.50-v7+ #1122 SMP Tue Jun 19 12:26:26 BST 2018 armv7l GNU/Linux\\
Linux orvavista 4.5.0-2-amd64 #1 SMP Debian 4.5.5-1 (2016-05-29) x86_64 GNU/Linux\\
```

2022: 5.15.61 (1) 5.15.0 (1) (ubuntu?) 5.15.32 (1) 5.4.51 (1)

- 6.c. Disk space. Why `-h`? Human readable. what does that mean? Why is it not the default? At least Linux defaults to 1kB blocks (UNIX was 512) Lots of large disks.

Raspberry Pi Booting

- Unusual – GPU handles it
- Small amount of firmware on SoC
- ARM chip brought up inactive (in reset)
- Videocore loads first stage from ROM

Raspberry Pi Booting (pre pi4)

- Videocore 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 onto ARM chip `kernel.img`. (also reads some config files there first) (4M)

Pi4 booting

- `https://www.raspberrypi.org/documentation/hardware/raspberrypi/booteeprom.md`
- SPI EEPROM holds equivelent of `bootcode.bin`, no longer read from partition
- Why? SDRAM, PCIe USB, etc are more complex
- Supports network and USB booting which is much more complex than just loading a file off of SD card

Typical ARM booting

- The UBoot bootloader is common
- ARM chip runs first-stage boot loader (often MLO)
- Then loads second-stage (uboot)

Why a FAT Partition?

- /boot on Pi is a legacy (40+ years old) File-Allocation Table (FAT) filesystem
- 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

Boot Methods

- 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 (systemd?)

- 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 (ld.so)
- 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.

Shared vs Staic Libraries

- Shared libraries, only need one copy of code on disk and in memory
  - Good for embedded system (less room needed)
  - Good for security updates (only need to update lib, not every program using it
- Static libraries, all libraries included
  - No dependencies
- These days maybe containers, docker, kubertenes
- Can use `ldd` to view library usage

Viewing Processes

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