ECE 471 – Embedded Systems
Lecture 17

Vince Weaver

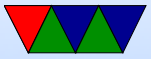
http://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

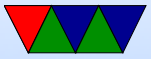
8 October 2021

Announcements

- Project info coming soon.
- ECE Seminar today at 2pm in Hill Auditorium, attend it if you can
- HW#5 was due
- HW#6 might be delayed, in any case will be due in *two* weeks
- Midterm is a week from Friday, the 15th



- Review on Wed
- No class Monday (Fall Break)

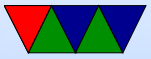


Device Detection

- 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.

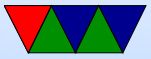
- Old ARM, hard-coded. So a rasp-pi kernel only could boot on Rasp-pi. Lots of pound-defined and hard-coded hw info.

- 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.

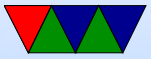
- Does mean that updating to a new kernel can be a pain.



Detecting Devices

There are many ways to detect devices

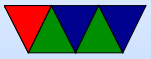
- Guessing/Probing – can be bad if you guess wrong and the hardware reacts poorly to having unexpected data sent to it
- Standards – always knowing that, say, VGA is at address 0xa0000. PCs get by with defacto standards
- Enumerable hardware – busses like USB and PCI allow you to query hardware to find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



it is located

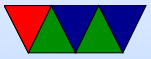
- Hard-coding – have a separate kernel for each possible board, with the locations of devices hard-coded in. Not very maintainable in the long run.

- Device Trees – see next slide



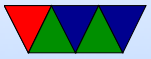
Devicetree

- 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



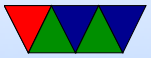
to describe all the hardware available. Thus kernel much more generic

- Still working on issues with this.

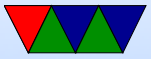


Booting Linux

- 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?)

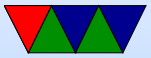


- Run init scripts
- rc boot scripts, /etc/rc.local
  Start servers, or “daemons” as they’re called under Linux.
- fork()/exec(), run login, run shell



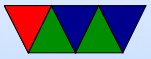
How a Program is Loaded on Linux

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



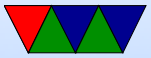
to entry address (specified by executable)

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



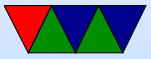
Viewing Processes

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

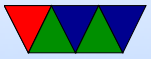


Homework 6 – Background

- You have two weeks to do this one
- Handout should cover most of it
- bit-banging i2c
- Why not bitbang everything? A pain. Hardware does it for you. Hardware even does more, can often buffer or DMA, timing more exact.
- Why might you want to bitbang i2c? Only have one i2c bus? Or no i2c bus, only GPIOs? kernel has bitbang driver

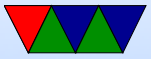


- Tell my boring frontpath i2c-bitbang story

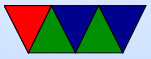


Homework 6 – Implementation

- Use the gpio interface to drive the SDA and SCL lines to manually run the 4x7 LED display
- Still easier than full bitbang, where you'd have to write to various i/o addresses
- A lot of the code is provided for you, follow the directions
- How do you set SDA low?
  Set output to '0'
- How do you set SDA high?
  Open collector, need to let it float, not be driven 1



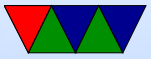
The Linux interface lets you select a line to be open-drain and when you do that, when you output a 1 it knows to switch the pin to “input” which lets the line float



Homework 6 – Multiple Files

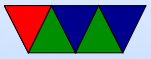
- This homework has the compiler compiling multiple small files and then linking them together.
- Why do this? Easier to edit smaller files, easier if collaborating with others, lets you share code without cut and pasting
- `gcc -c file.c` creates `file.o`
- Link a number of files together

  `gcc -o executable file1.o file2.o file3.o`
- This is how you create libraries (static just a matter of



`ar`, dynamic are a bit more complicated)

- All global functions/vars are exported, unless you declare them `static`
- How do you know how to call functions in other files? Need to pre-declare prototype so the compiler knows how to set up the registers before calling. Traditionally with C this is done in a .h header file
- Include files with `#include "file.h"`. You may also have seen angle brackets, what is the difference?
- By default all functions/global vars are exported. How can you specify to only be visible in own file? Use the



`static` keyword.