ECE 598 – Advanced Operating Systems
Lecture 2

Vince Weaver

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

vincent.weaver@maine.edu

15 January 2015

Announcements

- Update on room situation
  (shouldn't be locked anymore, but not much chance of getting a better room)

Building Linux by Hand

- Check out with git or download tarball:

  `git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git`

  `http://www.kernel.org/pub/linux/kernel/v3.x/`

- Configure.

  `make config` or `make menuconfig`

  Also can copy existing `.config` and run `make oldconfig`

- Compile.

`make`

What does `make -j 8` do differently?

- Make the modules.

  `make modules`

- `sudo make modules_install`

- copy bzImage to boot, update boot loader

Building Linux Automated

- If in a distro there are other commands to building a package.
- For example on Debian `make-kpkg --initrd --rootcmd fakeroot kernel_image`
- Then `dpkg -i` to install; easier to track

Overhead (i.e. why not to do it natively on a Pi)

- Size – clean git source tree (x86) 1.8GB, compiled with kernel, 2.5GB, compiled kernel with debug features (x86), 12GB!!!
  Tarball version with compiled kernel (ARM) 1.5GB
- Time to compile – minutes (on fast multicore x86 machine) to hours (18 hours or so on Pi)

Linux on the Pi

- Mainline kernel, bcm2835 tree
  Missing some features

- Raspberry-pi foundation bcm2708 tree
  More complete, not upstream

Compiling – how does it work?

- Take C-code
- Turn into assembler
- Run assembler to make object (machine language files). Final addresses not necessarily put into place
- Run linker (which uses linker script)
- Makes some sort of executable.

- How do you get first compiler written? Assembly, cross-compiler, native compiler

Tools

- compiler: takes code, usually (but not always) generates assembly
- assembler: GNU Assembler as (others: tasm, nasm, masm, etc.)
  creates object files
- linker: ld
  creates executable files. resolves addresses of symbols. shared libraries.

Converting Assembly to Machine Language

Thankfully the assembler does this for you.

ARM32 ADD instruction – `0xe0803080` == `add r3, r0, r0, lsl #1`

`ADD{S}<c> <Rd>,<Rn>,<Rm>{,<shift>}`

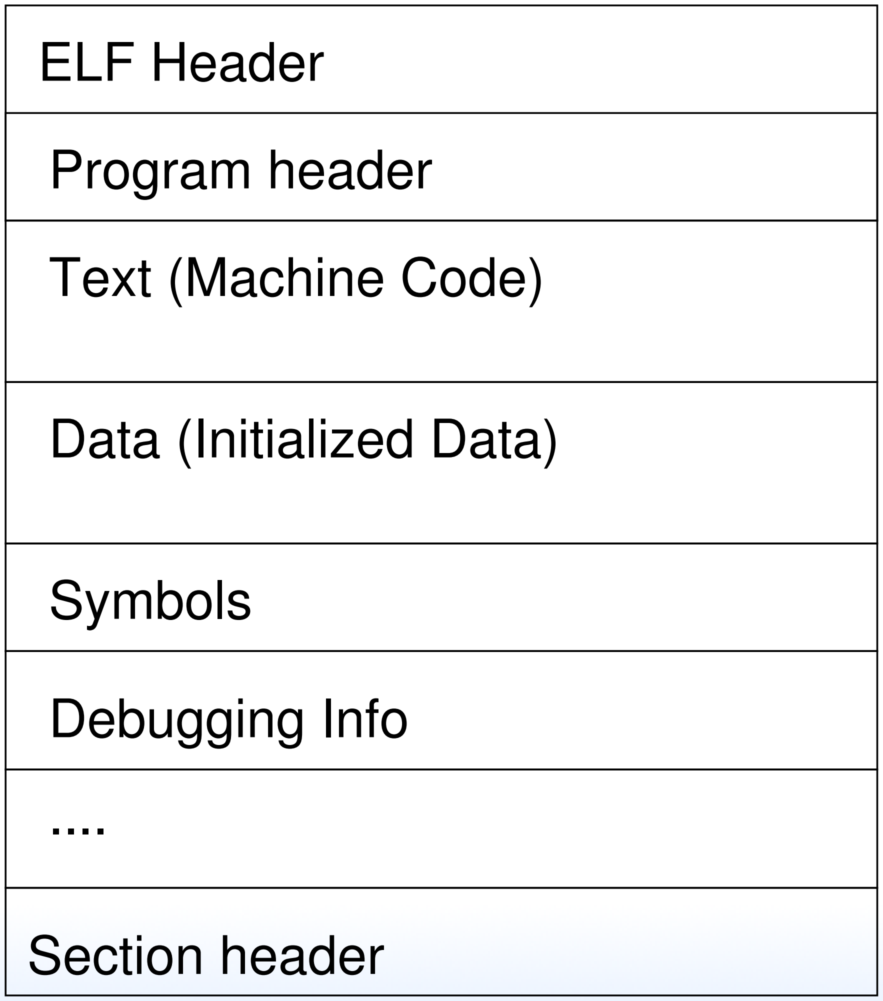
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
cond				0	0	0	0	1	0	0	S	Rn			
								Opcode							

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Rd				Shift imm5					Shift typ		Sh Reg	Rm			

Executable Format

- ELF (Executable and Linkable Format, Extensible Linking Format)
  Default for Linux and some other similar OSes
  header, then header table describing chunks and where they go
- Other executable formats: a.out, COFF, binary blob

ELF Layout


ELF Header
Program header
Text (Machine Code)
Data (Initialized Data)
Symbols
Debugging Info
....
Section header

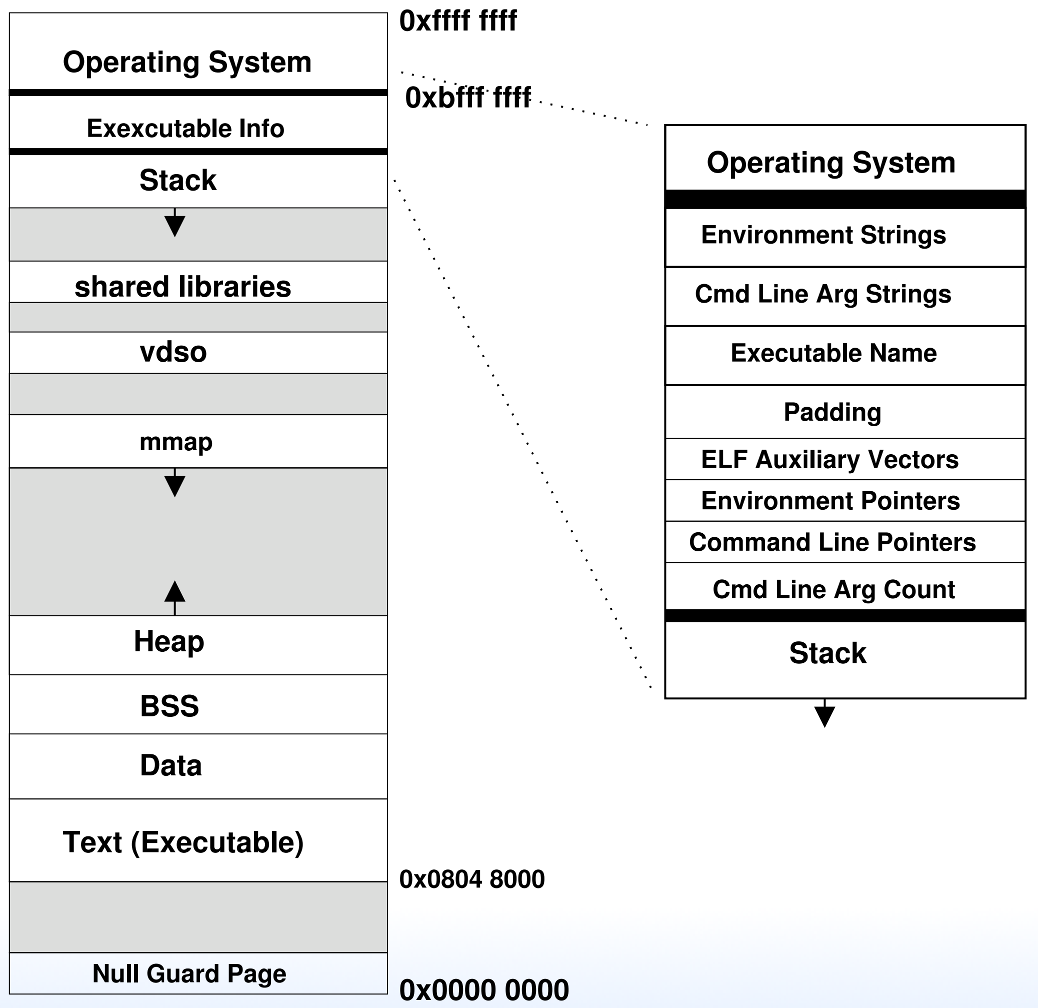
ELF Description

- ELF Header includes a “magic number” saying it’s 0x7f,ELF, architecture type, OS type, etc. Also location of program header and section header and entry point.
- Program Header, used for execution:
  has info telling the OS what parts to load, how, and where (address, permission, size, alignment)
- Program Data follows, describes data actually loaded into memory: machine code, initialized data

- Other data: things like symbol names, debugging info (DWARF), etc.
  DWARF backronym $=$ “Debugging with Attributed Record Formats”

- Section Header, used when linking:
  has info on the additional segments in code that aren’t loaded into memory, such as debugging, symbols, etc.

Linux Virtual Memory Map



Program Memory Layout on Linux

- Text: the program's raw machine code
- Data: Initialized data
- BSS: uninitialized data; on Linux this is all set to 0.
- Heap: dynamic memory. `malloc()` and `brk()`. Grows up
- Stack: LIFO memory structure. Grows down.

Program Layout

- Kernel: is mapped into top of address space, for performance reasons
- Command Line arguments, Environment, AUX vectors, etc., available above stack
- For security reasons “ASLR” (Address Space Layout Randomization) is often enabled. From run to run the exact addresses of all the sections is randomized, to make it harder for hackers to compromise your system.

Cross-compiling

- Building for a different architecture
- Why? ARM machines often slow
- Why not? Source tree has to be handle this. Makefile. etc. Usually easier to compile natively
- Linux kernel tends to cross compile OK.

Bootloaders on ARM

- uBoot – Universal Bootloader, for ARM and under embedded systems
- So both BIOS and bootloader like minimal OSes

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

Booting on x86

- BIOS
- Bootloader (let's say GRUB). Loads 512Byte bootsector to 0x7c00
- Loads second part
- Loads final part capable of booting Linux
- 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. Orginally 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. Id 64-bit, optionally switch to 64-bit

Decompressed kernel to 0x10 0000 (might have to move decompress code). (above 1MB. Why? 640k) Wht 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
- disk partitions, where bootloader stored, etc

Booting on typical ARM/uboot

- vmlinux. strip. compress. piggy / piggyback. decompression code tacked on convert to zImage. mkuimage converts to uimage suitable for booting with uboot
- No bios really. Bootloader provides all info.
- Modern day Device Tree provides config info for hardware (memory size, interrupts, what hardware is there)

More how setup

- Linker script
- describes sections, where they should go

More 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
- 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.

- This runs on videocard, enables SDRAM, then loads `start.elf`
- This initializes things, the loads and boots Linux `kernel.img`. (also reads some config files there first)

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

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?