Bootstrapping Embedded Linux

Josh Triplett

July 16, 2008

What does “bootstrapping” mean?

1. The entire boot process of a system, from application of power to performing its intended function. Usually just called “booting”.
2. Manually setting up a system to bootstrap itself in sense 1.

Overview

- Comparison of PCs versus embedded systems
- Walkthrough of the embedded boot process
- New technologies introduced for each step of bootstrapping
- Summary of embedded Linux bootstrap process
- Summary of embedded Linux boot process
- All information based on experiences with the PowerPC flight computer for the Portland State Aerospace Society rocket avionics system

PC boot process

- Power on

PC boot process

- Power on
- BIOS gets control

PC boot process

- Power on
- BIOS gets control
- BIOS initializes some hardware

PC boot process

- Power on
- BIOS gets control
- BIOS initializes some hardware
- BIOS loads bootloader

PC boot process

- Power on
- BIOS gets control
- BIOS initializes some hardware
- BIOS loads bootloader
- Bootloader loads operating system kernel

PC boot process

- Power on
- BIOS gets control
- BIOS initializes some hardware
- BIOS loads bootloader
- Bootloader loads operating system kernel
- Kernel probes hardware

PC boot process

- Power on
- BIOS gets control
- BIOS initializes some hardware
- BIOS loads bootloader
- Bootloader loads operating system kernel
- Kernel probes hardware
- Kernel finds and mounts root filesystem

PC boot process

- Power on
- BIOS gets control
- BIOS initializes some hardware
- BIOS loads bootloader
- Bootloader loads operating system kernel
- Kernel probes hardware
- Kernel finds and mounts root filesystem
- Kernel runs init

PC boot process

- Power on
- BIOS gets control
- BIOS initializes some hardware
- BIOS loads bootloader
- Bootloader loads operating system kernel
- Kernel probes hardware
- Kernel finds and mounts root filesystem
- Kernel runs init
- Init gets userspace up and running

How does an embedded system differ?

- No BIOS; initial control passes to bootloader

How does an embedded system differ?

- No BIOS; initial control passes to bootloader
- Limited options for storage

How does an embedded system differ?

- No BIOS; initial control passes to bootloader
- Limited options for storage
- Hardware normally doesn't change

How does an embedded system differ?

- No BIOS; initial control passes to bootloader
- Limited options for storage
- Hardware normally doesn't change
- Constraints on storage and memory; want minimal root filesystem and userspace

Embedded Technologies

- U-Boot, a powerful embedded bootloader
- Memory Technology Device (MTD), a flexible storage medium
- Compiled device trees, for static hardware configuration
- initramfs, for the root filesystem
- BusyBox, for a lightweight userspace

U-Boot

- BIOS, bootloader, diagnostic tool
- Very popular for embedded systems
- Stored in flash; booted from flash
- Upgradable, but need a JTAG if something goes wrong
- Not generic; compiled for specific system board
- Portable to most architectures
- Drivers for various hardware needed to boot
- Uses `uImage` kernel format;
  `apt-get install uboot-mkimage`
- Highly scriptable

U-Boot scripting

- Accessible via serial or network
- Can run commands interactively or automatically
- U-Boot commands can:
  - Read/write MTD flash, memory, disk filesystems
  - Use the network: IP configuration, telnet server, ping, tftp
  - Probe hardware or perform diagnostics
  - Boot kernels
- "Environment" system of key-value pairs
- Environment variables can contain U-Boot commands
- U-Boot runs the environment variable `bootcmd` on boot
- Environment transient unless explicitly saved
- Save environment to flash when working as desired

Memory Technology Devices (MTD)

- Memory-mapped flash device
- Non-volatile
- Accessible from U-Boot and Linux
- Partitioned by convention. Regions for U-Boot, U-Boot environment, kernel, device tree, filesystem
- Use U-Boot to load kernel or other file from TFTP into RAM, write to flash, and load from flash on future boots

Compiled device trees, for static hardware

- Embedded hardware typically doesn't change
- Embedded hardware often not probed, or even probe-able
- Compile a static device tree with hardware information
- Compiled from a text description in device-tree source (dts) into a format similar to Open Firmware
- `apt-get install device-tree-compiler`
- Also contains configuration information, such as MTD partitioning or function of multi-function ports
- U-Boot passes the device-tree binary (dtb) or "flattened device tree" (fdt) to the kernel
- Need semi-recent U-Boot to use compiled device trees

Finding a root filesystem

- Can attach a disk or CF/SD card, and use that as root.
- Can use MTD flash as root
  - Read/write with flash filesystem like JFFS2
- Can store initramfs in flash and use as root filesystem

initrd and initramfs: background

- root on disk:

initrd and initramfs: background

- root on disk: easy enough to deal with in kernel

initrd and initramfs: background

- root on disk: easy enough to deal with in kernel
- root on NFS:

initrd and initramfs: background

- root on disk: easy enough to deal with in kernel
- root on NFS: possible, but need network configuration and NFS client in kernel

initrd and initramfs: background

- root on disk: easy enough to deal with in kernel
- root on NFS: possible, but need network configuration and NFS client in kernel
- root on LVM:

initrd and initramfs: background

- root on disk: easy enough to deal with in kernel
- root on NFS: possible, but need network configuration and NFS client in kernel
- root on LVM: too much configuration to handle in kernel

initrd and initramfs: background

- root on disk: easy enough to deal with in kernel
- root on NFS: possible, but need network configuration and NFS client in kernel
- root on LVM: too much configuration to handle in kernel
- root on encrypted LVM on RAID with hibernation image on swap in the same LVM:

initrd and initramfs: background

- root on disk: easy enough to deal with in kernel
- root on NFS: possible, but need network configuration and NFS client in kernel
- root on LVM: too much configuration to handle in kernel
- root on encrypted LVM on RAID with hibernation image on swap in the same LVM: Right Out

initrd and initramfs: background

- root on disk: easy enough to deal with in kernel
- root on NFS: possible, but need network configuration and NFS client in kernel
- root on LVM: too much configuration to handle in kernel
- root on encrypted LVM on RAID with hibernation image on swap in the same LVM: Right Out
- Also used in generic non-embedded GNU/Linux distributions, who want to build everything as modules, including disk and filesystem drivers

initrd and initramfs: background

- root on disk: easy enough to deal with in kernel
- root on NFS: possible, but need network configuration and NFS client in kernel
- root on LVM: too much configuration to handle in kernel
- root on encrypted LVM on RAID with hibernation image on swap in the same LVM: Right Out
- Also used in generic non-embedded GNU/Linux distributions, who want to build everything as modules, including disk and filesystem drivers
- Need userspace before root filesystem available

initrd and initramfs: background

- root on disk: easy enough to deal with in kernel
- root on NFS: possible, but need network configuration and NFS client in kernel
- root on LVM: too much configuration to handle in kernel
- root on encrypted LVM on RAID with hibernation image on swap in the same LVM: Right Out
- Also used in generic non-embedded GNU/Linux distributions, who want to build everything as modules, including disk and filesystem drivers
- Need userspace before root filesystem available
- Bootloader loads initrd/initramfs into memory and provides address to kernel

initrd - the old solution

- initial ram disk
- Disk image with filesystem
- Kernel mounts initrd and runs `/linuxrc`
- `/linuxrc` must make real root device available, then return; kernel then runs /sbin/init from new root
- Inflexible: invoked in the middle of the in-kernel algorithm to find the root filesystem
- Inefficient: in-RAM block device "cached" elsewhere in RAM

initramfs - the new hotness

- gzipped cpio archive
- Extracted into a tmpfs - Linux disk cache used as filesystem
- Kernel runs `init`, userspace does the rest
- Can mount a filesystem and exec another init
- Can provide all necessary functionality itself

BusyBox - lightweight userspace

- A standard GNU/Linux userspace takes up a lot of space
- Busybox provides replacements for many useful programs
  - Essential programs: init, coreutils, shells, ...
  - Extra services: httpd, dpkg, udhcpc, ...
- All programs in one binary for maximum code sharing
- busybox checks `argv[0]`; just link it to many different paths
- Configurable via Kconfig, like the Linux kernel
- Fine-grained choices:
  - Which commands do you need?
  - Which command-line options do you need?
  - What functionality do you need?

Summary of embedded bootstrapping

- Build cross-compiler for target architecture (PowerPC)

Summary of embedded bootstrapping

- Build cross-compiler for target architecture (PowerPC)
- Build current U-Boot for the target architecture

Summary of embedded bootstrapping

- Build cross-compiler for target architecture (PowerPC)
- Build current U-Boot for the target architecture
- Flash new U-Boot using old U-Boot, and hope it boots (it did)

Summary of embedded bootstrapping

- Build cross-compiler for target architecture (PowerPC)
- Build current U-Boot for the target architecture
- Flash new U-Boot using old U-Boot, and hope it boots (it did)
- Cross-compile Linux kernel as a uImage (need `mkimage`)

Summary of embedded bootstrapping

- Build cross-compiler for target architecture (PowerPC)
- Build current U-Boot for the target architecture
- Flash new U-Boot using old U-Boot, and hope it boots (it did)
- Cross-compile Linux kernel as a uImage (need `mkimage`)
- Configure and cross-compile BusyBox

Summary of embedded bootstrapping

- Build cross-compiler for target architecture (PowerPC)
- Build current U-Boot for the target architecture
- Flash new U-Boot using old U-Boot, and hope it boots (it did)
- Cross-compile Linux kernel as a uImage (need `mkimage`)
- Configure and cross-compile BusyBox
- Build the initramfs from BusyBox and some config files

Summary of embedded bootstrapping

- Build cross-compiler for target architecture (PowerPC)
- Build current U-Boot for the target architecture
- Flash new U-Boot using old U-Boot, and hope it boots (it did)
- Cross-compile Linux kernel as a uImage (need `mkimage`)
- Configure and cross-compile BusyBox
- Build the initramfs from BusyBox and some config files
- Copy and customize device-tree source template for our board

Summary of embedded bootstrapping

- Build cross-compiler for target architecture (PowerPC)
- Build current U-Boot for the target architecture
- Flash new U-Boot using old U-Boot, and hope it boots (it did)
- Cross-compile Linux kernel as a uImage (need `mkimage`)
- Configure and cross-compile BusyBox
- Build the initramfs from BusyBox and some config files
- Copy and customize device-tree source template for our board
- Compile device-tree source with `device-tree-compiler`

Summary of embedded bootstrapping

- Build cross-compiler for target architecture (PowerPC)
- Build current U-Boot for the target architecture
- Flash new U-Boot using old U-Boot, and hope it boots (it did)
- Cross-compile Linux kernel as a uImage (need `mkimage`)
- Configure and cross-compile BusyBox
- Build the initramfs from BusyBox and some config files
- Copy and customize device-tree source template for our board
- Compile device-tree source with `device-tree-compiler`
- Use U-Boot and TFTP to copy kernel, initramfs, and dtb to RAM, and flash them to appropriate addresses in MTD flash

Summary of embedded bootstrapping

- Build cross-compiler for target architecture (PowerPC)
- Build current U-Boot for the target architecture
- Flash new U-Boot using old U-Boot, and hope it boots (it did)
- Cross-compile Linux kernel as a uImage (need `mkimage`)
- Configure and cross-compile BusyBox
- Build the initramfs from BusyBox and some config files
- Copy and customize device-tree source template for our board
- Compile device-tree source with `device-tree-compiler`
- Use U-Boot and TFTP to copy kernel, initramfs, and dtb to RAM, and flash them to appropriate addresses in MTD flash
- Configure U-Boot `bootcmd` to boot Linux from MTD flash

Summary of embedded boot process

- Power on

Summary of embedded boot process

- Power on
- U-Boot gets control

Summary of embedded boot process

- Power on
- U-Boot gets control
- U-Boot initializes minimum necessary hardware

Summary of embedded boot process

- Power on
- U-Boot gets control
- U-Boot initializes minimum necessary hardware
- U-Boot runs environment variable “bootcmd” on boot

Summary of embedded boot process

- Power on
- U-Boot gets control
- U-Boot initializes minimum necessary hardware
- U-Boot runs environment variable “bootcmd” on boot
- bootcmd tells U-Boot to boot Linux from flash with initramfs and dtb

Summary of embedded boot process

- Power on
- U-Boot gets control
- U-Boot initializes minimum necessary hardware
- U-Boot runs environment variable “bootcmd” on boot
- bootcmd tells U-Boot to boot Linux from flash with initramfs and dtb
- U-Boot loads Linux, initramfs, and dtb from flash into RAM

Summary of embedded boot process

- Power on
- U-Boot gets control
- U-Boot initializes minimum necessary hardware
- U-Boot runs environment variable “bootcmd” on boot
- bootcmd tells U-Boot to boot Linux from flash with initramfs and dtb
- U-Boot loads Linux, initramfs, and dtb from flash into RAM
- bootcmd boots Linux, passing initramfs and dtb

Summary of embedded boot process

- Power on
- U-Boot gets control
- U-Boot initializes minimum necessary hardware
- U-Boot runs environment variable “bootcmd” on boot
- bootcmd tells U-Boot to boot Linux from flash with initramfs and dtb
- U-Boot loads Linux, initramfs, and dtb from flash into RAM
- bootcmd boots Linux, passing initramfs and dtb
- Kernel finds hardware via dtb

Summary of embedded boot process

- Power on
- U-Boot gets control
- U-Boot initializes minimum necessary hardware
- U-Boot runs environment variable “bootcmd” on boot
- bootcmd tells U-Boot to boot Linux from flash with initramfs and dtb
- U-Boot loads Linux, initramfs, and dtb from flash into RAM
- bootcmd boots Linux, passing initramfs and dtb
- Kernel finds hardware via dtb
- Kernel extracts initramfs into tmpfs

Summary of embedded boot process

- Power on
- U-Boot gets control
- U-Boot initializes minimum necessary hardware
- U-Boot runs environment variable “bootcmd” on boot
- bootcmd tells U-Boot to boot Linux from flash with initramfs and dtb
- U-Boot loads Linux, initramfs, and dtb from flash into RAM
- bootcmd boots Linux, passing initramfs and dtb
- Kernel finds hardware via dtb
- Kernel extracts initramfs into tmpfs
- Kernel runs init

Summary of embedded boot process

- Power on
- U-Boot gets control
- U-Boot initializes minimum necessary hardware
- U-Boot runs environment variable “bootcmd” on boot
- bootcmd tells U-Boot to boot Linux from flash with initramfs and dtb
- U-Boot loads Linux, initramfs, and dtb from flash into RAM
- bootcmd boots Linux, passing initramfs and dtb
- Kernel finds hardware via dtb
- Kernel extracts initramfs into tmpfs
- Kernel runs init
- Init gets userspace up and running