Power button to login prompt

or how to lose your sanity and sleep over thirty seconds

Greg Fefelov, gfv@zoidberg.md

March 29, 2016

Chapter 1

where the world isn't limited to x86 even in x86-based systems

Power button to first x86 code

Code and microprocessors

wait, I've thought one architecture was enough?

- Embedded Controller firmware[1]
- Intel® Management Engine firmware[1]
- BIOS/EFI[1]
- External device ROMs
- ...and finally
- Code on disks

[1] Physically in an SPI flash on the motherboard.

Who the hell is an Embedded Controller?

sounds fishy

- Tiny separate controller with an obscure architecture[1]
- Crippled-down ISA bus to the chipset
- Lots of GPIO pins
- Direct links to power controller
- Thermal management, fan speed controls, etc..
- Keyboard proxy

[1] manufacturer-specific, Lenovo uses Renesas H8 chips

What's up with Intel® Management Engine?

you know shit gets serious when there are trademark signs

- Physically on your (recent) Intel processor's package
- Yet separate from x86 cores
- Boot code is physically in processor package
- Has even more obscure architecture
- Direct access to all your memory
- A trusted component of security subsystem
- Remote management

All that stuff is encrypted, so you can't read it, you can't write it :(

External ROMs

wait, there is more of this?

- All over your devices
- Netboot, PXE
- Graphics initialization

How all this stuff works

and now we'll try to take off with all this crap!

- EC wakes up
- Checks battery and AC status
- Set up its own internal tasks like watchdogs..
- Toggles RESET on x86 unit
- Management Engine loads code from flash and starts to run
- External ROMs are not run just yet

Chapter 2

where nothing is given and bootstraps are too short

First x86 code to boot loader

Baby steps

a 16nm baby with 500-picosecond steps

- As stated in Processor starts executing in real mode from memory address 0xfffffff0
- There are two WTFs here.

How Can 0xfffffff0 Be Real If Our Memory Isn't Real?

or: How I Learned to Stop Worrying and Love the Kludge

- Not all RAM is actually RAM
- Some parts of RAM address space are actually SPI ROM
- Some parts of RAM address space are actually PCI I/O
- Routed by Southbridge
- Caches have to be explicitly enabled

Unreal memory for real men

0xffffffffuuuuuuuuuuuu

- Processors can get away with any violations if they set the rules
- Even accessing 4G of memory in real mode
- Processor sets CS to have base 0xffff0000 limit 0xffff and IP 0xfff0

The First Great Choice

Common tasks for early init

you have to start with hello

- Set up memory maps
- Initialize selected hardware
- Find a bootable medium
- Load OS boot loader

BIOS overview

jurassic era

- 1970s technology
- Developed for IBM PC model 5150, used ever since
- Never leaves real mode
- Launches VideoBIOS and code NIC ROM
- Has limited knowledge of modern technology
- Provides real-mode interrupt-based API for basic IO[2]

[2] so generally useless unless you're Marty McFly

BIOS launch process

rawr i'm a tiger

- Initialize memory
- Run VideoBIOS
- Scan storage devices
- Show a pretty splash screen
- Find first device with 0x55,0xaa in last positions of first sector
- Load first sector to memory, offset 0x7c00
- Jump there
- ???
- Profit!

EFI overview

paleogene era

- 1990s technology
- Started from Itanium, now in AMD64, ARM etc.
- Has concept of disk partitions and FAT[3]
- Modular, can run special executables from system partitions
- Has drivers!

[3] actually defines on-disk FAT structures and partition table layout in specification

EFI launch process

meow i'm a cat

- Set up temporary RAM
- Initialize CPU, chipset, memory
- Load drivers
- Select boot device
- Start /EFI/Boot/BOOTX64.efi[4] or whatever specified

[4] other platforms have different default paths; 32-bit systems have BOOTX32.efi, ARM uses BOOTARM.efi

Chapter 3

where you have your sanity and a choice

Boot loader to /sbin/init

A basic MBR loader

basic usually means useless

- Reads partition table from itself
- Finds partition with active flag
- Moves itself from 0x7c00
- Reads first sector to 0x7c00, using BIOS services
- Jumps there

How many BIOS bootloaders are there?

this question opens a can of worms

- GRUB 0.99
- GRUB2
- LILO ♡
- Syslinux
- NTLDR

How it actually works with GRUB

fractals all the way down

- Modern boot loaders are smart
- Smart means fat, they don't really fit in 512 bytes
- So GRUB2 uses several sectors
- With Reed-Solomon coding
- Recognizes /boot file system
- Loads menu and other stuff from there
- Transitions into protected mode with identity $V \leftrightarrow P$ mapping

EFI boot loader

EFI is love, EFI is life

- Is in system native mode (32 or 64 bit) from early stages
- Executables are PE images
- Started by firmware from EFI system partition
- Provides rich API for disk I/O, graphics I/O
- (Almost) no size limits, no Reed-Solomon trickery
- Firmware lets you actually choose what executable to launch
- Linux kernel can be compiled as an EFI[5]
- Therefore no need for a separate boot loader

[5] fun fact: contains MZ header, so can be (sort of) run from DOS

Linux kernel init

we still haven't got to this? damn we are slow

- Decompress the kernel
- Set up basic system primitives like locks, rudimentary memory primitives
- Print linux banner
- Set up first processor
- Make RAM space for initrd[6] (if necessary)
- Load memory maps (with e820 or EFI)
- Init memory management
- Read SMP info from APIC, set things up, launch CPUs, NUMA..
- Set up scheduler, RCU[7]
- Scan buses, set up devices

[6] will be discussed later

[7] read-copy-update

Final transition

oh god i'm so bored somebody shoot me

```
if (ramdisk_execute_command) {
        ret = run_init_process(ramdisk_execute_command);
        if (!ret)
                return 0;
}
if (execute_command) {
        ret = run_init_process(execute_command);
        if (!ret)
                return 0;
}
if (!try_to_run_init_process("/sbin/init") ||
        !try_to_run_init_process("/etc/init") ||
        !try_to_run_init_process("/bin/init") ||
        !try_to_run_init_process("/bin/sh"))
                return 0;
```

Simple is too simple

what we've got here is failure to communicate

- / is on a device or FS with driver built as kernel module
- / filesystem is encrypted
- / filesystem is in LVM
- / is on a remote server
- You want to adjust the system before starting real init
- Lots of actual use cases!

Quite early userspace

there's an app for that

- Before mounting the real FS, use a small filesystem with just what's needed to get going
- Set up the system from this pre-environment
- Then switch to the real FS with a real init
- initrd, initramfs

initrd, initramfs

come on, there are too many choices!

- initrd is actually a file system image
- initramfs is an archive of a file system decompressed into ramdisk
- Ramdisk is neither a file system nor a device, it just kinda exists
- Therefore they use different methods to switch file systems
- initrd uses `pivot_root`
- initramfs just mounts the true root over / and then chroots
- kernel automounts /proc in initramfs; initrd has to do it manually

pivot_root

this syscall has no libc condom, this is the only slide where I will announce the subtitle

- File system A mounted at / and a file system B mounted at /another
- `pivot_root("/another, "/old")` will make B the new root, and A will be mounted at /old in B.
- Every process with working directory in A root will now have cwd in B root
- ...except maybe it won't.
- Quote from man 2 pivot_root:

  pivot_root() may or may not change the current root and the current working directory of any processes or threads which use the old root directory.

- It does though[8]. Quote from fs/namespace.c around line 2864, Linux 3.18-rc5

  sets root/cwd of all processes which had them on the current root to new_root.

[8] you didn't hear it from me, this is an implementation detail

What should initrd or initramfs do?

what's in the box?!

- An image of a file system
- Usually has just enough files to load real file system
- Mounts sysfs and procfs
- Reads parameters from kernel command line[9]
- Launches udev[10]
- Waits for devices to settle
- Mounts real root
- Switches and runs init with an appropriate tool (pivot_root[11] in initrd or run-init in initramfs)

---

[9] literally `cat /proc/cmdline`

[10] will be discussed later

[11] not a syscall! it's a binary wrapper for a kernel function

Chapter 4

where everything spirals out of control

/sbin/init to login prompt

Where do we go now?

<obscene joke cut out>

- We probably want to start services like login prompt, SSH, etc..
- We'd like to automagically enable all devices not discovered at boot
- Let's start from the end and see if anything works out

Hot plugging

some like it hot

- Plugging in an USB modem should work seamlessly: you should get a device node in /dev, it should be chowned to proper group
- Plugging in a memory stick should make it work automatically
- Shutting down an Ethernet card should stop all daemons bound to it
- You'd want some general implementation of this, right?

What is udev?

who devises these stupid names?

- Udev is a mature daemon
- Has kernel-side and userspace code
- Kernel part sends 'uevents' on any hardware state change to userspace
- Userspace part may act on them, according to user-supplied rules like these:
  ```
  SUBSYSTEM=="tty", KERNEL=="tty[1-6]",
  RUN+="console-setup-tty /dev/%k"
  ```
- It maintains a database with all plugged hardware
- Third-party software can query it or subscribe to events with libudev

What udev isn't

screwdev

- Not a silver bullet
- Not a coffee machine
- Not an interpreter for a Turing-complete language[12]

[12] okay, I lied here, but for your own good

A fork() in the road

interstate 60, anyone?

- Several solutions for running daemons
- SysVinit
- Upstart
- systemd

Things in common

this is a very short list

- PID 1 gets all processes with dead parents
- When processes don't get waitpid()ed, they linger as zombies
- A good pid 1 will listen to these events
- When pid 1 exits, kernel panics

SysVinit

V is a roman numeral for '5', it is that ancient

- A child of 1980s
- Governed by simple shell scripts
- Has concept of runlevels
- A close relative of slightly more recent BSD init
- No modern major Linux distribution uses it

Simple start

run, Forrest, run

- /sbin/init reads /etc/inittab
- /etc/inittab contains a list of scripts to run on runlevel change as well as a default runlevel
- Runlevel 0 is halt, rl 1 is single user, rl 2-5 are multi-user (2 is the default), rl 6 is reboot
- All runlevel changes run /etc/init.d/rc script with new runlevel as a parameter
- All init scripts for services are in /etc/init.d/
- To enable a service, place a symlink in /etc/rc$RUNLEVEL.d
- /etc/init.d/rc runs them sequentially
- Kludges: rudimentary dependency system based on comments, limited parallel launch

Upstart

rolling stones' start me up had had no influence over this

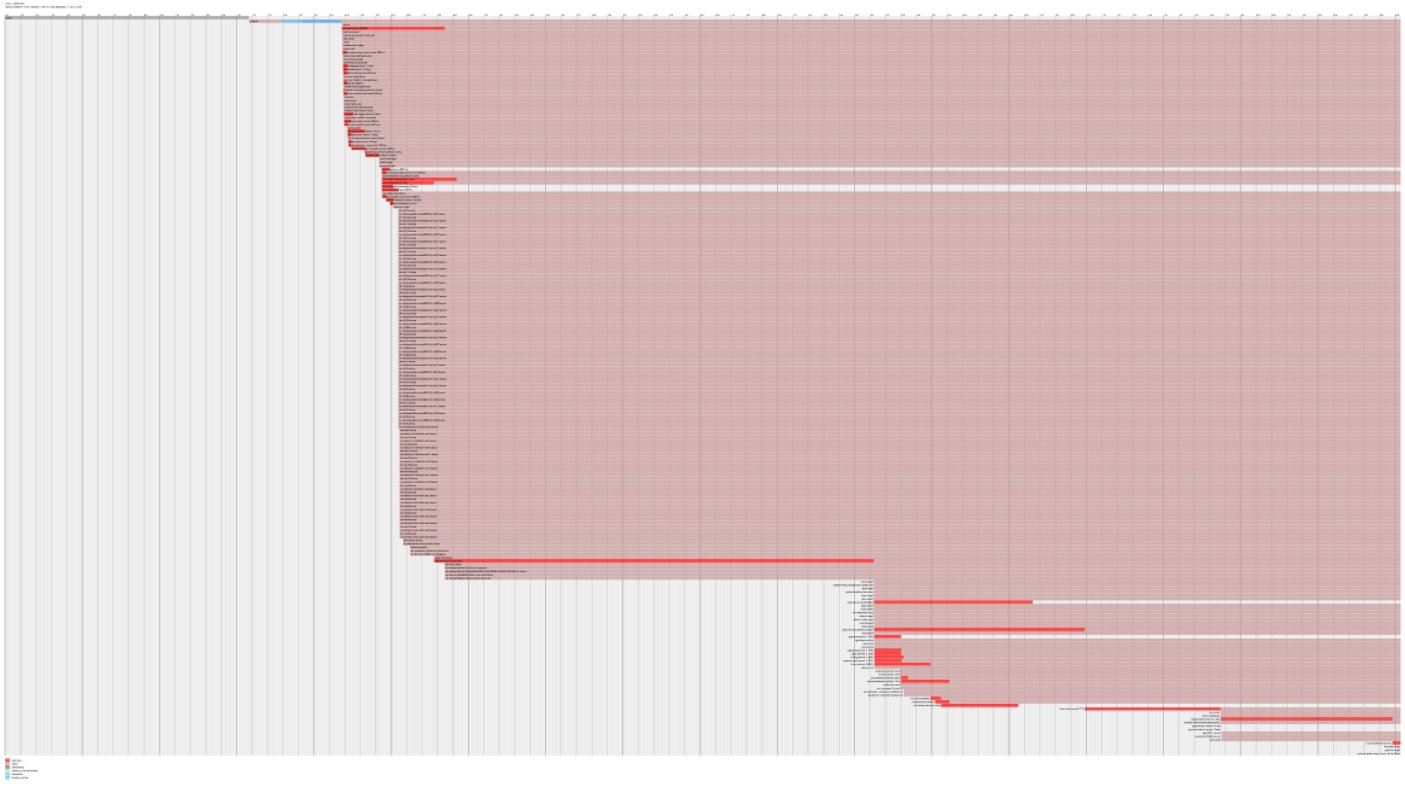
- Introduced in Ubuntu 6.10
- Never really took off
- Quietly substituted with systemd in recent Ubuntu releases
- Has event-based system: each service generates events on start and stop; events may trigger another service
- Can supervise services
- Supports proper parallel launch


SYSTEMD
SYSTEMD

systemd

D is roman 500, therefore it's 100 times better than sysvinit

- Inspired by OS X's launchd, released in 2010
- Attempts to manage everything from NTP to laptop lid behavior
- Contains udev as a subsystem
- Thinks in terms of units that depend, require or sequenced after each other
- Can generate pretty pictures:



systemd units

who needs reasons when you've got systemd?

- Services: simple, forking, oneshot, dbus, notify, idle
- Generators
- Targets
- Socket activation (a modern inetd)
- Device-based activation
- Device units are generated automagically based on udev database[13]
- FS units from fstab

[13] by default all block and network devices, and a few others

Launching order

dog says 'woof', cat says 'meow', systemd says 'failed'

- Generators
- Makes dependency tree
- Activation is split into two parts: gathering dependencies and launching
- Activates /lib/systemd/system/default.target
- Then recursively descends:
  - If activating unit A Requires B, then B is also set to activated, and A will be deactivated if B fails
  - If activating unit A Wants B, then B is also set to activated
  - If activating unit A After B, then A will proceed only after complete activation of B, or if B is not activated at all
- A lot of other relationships between units: PartOf, Requisite, PropagatesReloadTo...
- Inverse relationships: RequiredBy, WantedBy, Before...

How does it actually start SSH?

asking the right questions is the first step to right answers

- default.target is symlinked to graphical.target
- graphical.target Requires multi-user.target
- multi-user.target WantedBy ssh.service
- ssh.service Requires basic.target
- basic.target Requires sysinit.target
- sysinit.target Requires or Wants a lot of units that are started with the system

Epilogue

where audience has an epiphany

All software is shit. Hardware is primarily shit as well.
System administration is an art of staying afloat in the brown sea.
So it goes.

Credits

Thank you for coming!

@feelthefrog
vk.com/feelthefrog
gfv@zoidberg.md