Red Hat
In-Vehicle
Operating System

Eric Chanudet
Principal Software Engineer

Brian Masney
Principal Software Engineer

Improving Linux boot time for Automotive use-cases

June 2025

Observability: What's going on?

- Scope
  - From when, for how long?
- Feedback
  - How do we know?

Scope: When to start?

- Platforms may not cold start from the application CPU;
- Firmware and bootloaders (UEFI or not) take time before starting the kernel;
- Hypervisors anyone?
- Linux starts accounting for time late (after *time_init()*).

Scope: When to stop?

- What is a complete boot?
- What devices need to be running?
- What software needs to be operational?
- What software and devices need to be booted first?

Feedback: Tools and techniques

- Arch counters (cntvct/pct, tscs, etc)
- Kernel logs & system journal
- Kernel trace
- *systemd-analyze*
- *bootchard*
- *initcall_debug* or relevant trace events

Kernel

- Before booting
  - Configuration
- Kernel initialization
  - Before time and early setup
- Drivers
  - Tactical initialization

Before booting

- Tailored kernel configuration
  - Avoid unnecessary initcalls
  - Reduce binary/image size
- Quiet(er) logs
  - Great way to skew measurements (e.g, serial logs)
- Disable/Quiesce boot checks without recourse
  - Crypto selftests can add several hundred milliseconds (FIPS)
  - W+X mappings (*CONFIG_DEBUG_WX*)

Kernel initialization: Early blindspot

- Measurements often rely on *sched_clock()* (printk timestamps)
  - Initialized at *start_kernel()* -> *time_init()*
- Revealed silent delays early on that were optimised:
  - Arm64 linear-map creation had over zealous memory barriers and TLB invalidations
  - 32GB RAM platform went from ~350ms -> 25ms
- The blindspot is still there, can we reduce it?
  - Use an arch counter as *sched_clock()* earlier? Or with an epoch at *primary_entry*?
  - Inherit measurements passed by the firmware?
    - ACPI FPDT? ACPI is rare in embedded and infos recorded are limited;
    - DT bootstages? Only seen it in u-boot so far;
    - New stuff (e.g, using a reserved region and a Linux driver to read it)?

Kernel initialization: Optimizations

- Bottlenecks tend to be incremental. Consistent measurement is key.
- e.g, *struct page* initialization (*CONFIG_DEFERRED_STRUCT_PAGE_INIT*)
  - Deferred from *setup_arch()* to *page_alloc_init_late()*
  - In other words: From before *time_init()* to after kthreads are available.
  - Other bottlenecks may hide improvements (*early_pfn_to_nid()* redundant calls)
  - On QC Ride 4 (arm64), 32GB: ~230ms -> ~30ms
- Tracing early boot progression, even with ad-hoc code, is often useful to find these bottlenecks.

Drivers: Probe deferrals

- None is ideal :)
  - but they may not all be bottlenecks.
- fw_devlink, but not all dependencies can be inferred:
  - On Lenovo x13s, device_links_check_suppliers() device link loop doesn't prevent any
  - *initcall_debug* reveals more deferrals (total ~165ms using the following log)
- New ideas?

```
> journalctl --dmesg |
> sed -ne 's|.*probe of \([^ ]\+\) returned \(-*517\) after \([0-9]\+\).*|\1 \3|p' |
> awk '{ a[$1] += $2 } END { for (i in a) print i, a[i] }' |
> sort -k2 -h -r | head -n10
3250000.soundwire 32387
c440000.spmi:pmic@0:adc-tm@3400 29801
sound 26299
3210000.soundwire 23523
3330000.soundwire 15975
15000000.iommu 9854
aux_bridge.aux_bridge.1 9282
1c20000.pcie 4015
pmic_glink.altmode.0 2813
firmware:scm 2411
```

System initialization: RCU speed on RT kernels

- E.g, (repeated) *rcu_barrier()* can be costly especially on *PREEMPT_RT*
  - Left out *rcu_barrier()* after refactoring to a work-queue added ~100ms.
  - On QC Ride 4 (arm64), 32GB: ~100ms -> ~1ms
- mount/umount (MNT_DETACH), important during container startup
  - Hit a costly rcu_barrier() as well (failed attempt to by-pass it) (~25ms/syscall)
- RCU grace-periods in general can delay boot up.
  - We don't make real time guarantees during the boot phase: Use RCU expedited mode early on
  - Boot with: *rcupdate.rcu_normal_after_boot=0 rcupdate.rcu_expedited=1*
  - After boot, switch back to normal: *echo 1 > /sys/kernel/rcu_normal*
  - Quick tests on qemu shows that this can save about a half second of time booting to minimal userspace.

Drivers: Case-by-case optimizations

- The UFS host-controller negotiates the *Gear* (speed) it will use for its devices.
  - For Qualcomm boards, the firmware already configured the highest speed for the storage before the kernel.
    - The kernel re-negotiate the speed, this can be avoided.
    - On QC Ride 4, it saves ~50ms
- Redundant reset sequence of MDIO PHY device
  - Assert/deassert come with delays for completion that can be avoided.
  - This happens in multiple paths, but specifically when the device is probed.

To compress or not compress?

- If your kernel is small enough and your storage is fast enough, then booting with an uncompressed kernel may be faster.
- Same with the initramfs. We're currently using LZ4 compression for our initramfs due to space constraints for the combined kernel / initramfs.
- We are compressing our kernel modules on the root filesystem with zstd instead of xz.

Userland

- Initramfs
- udev and Systemd

systemd can be small enough for the initramfs

For distros that already use systemd, it can be also be made small enough to include in the initramfs from a boot speed perspective. We have a minimized systemd dracut module available at:

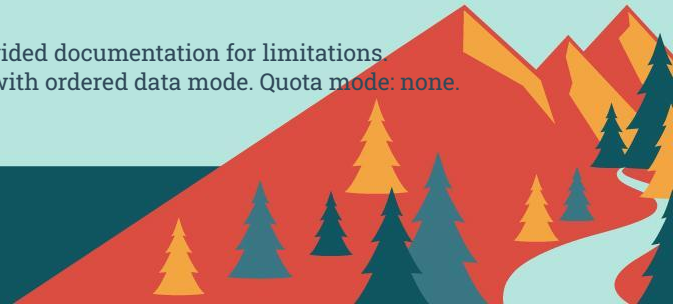
https://gitlab.com/CentOS/automotive/rpms/dracut-automotive

Note that by default this doesn't include journald by default, so you won't see any log output from the initramfs. This reduces the initramfs by ~130 KB.

```
localhost systemd[214]: mount-sysroot.service: Failed to connect stdout to the journal socket, ignoring: No such file or directory
```

Additionally include the 'systemd-journald' dracut module, and logging from the initramfs will appear:

```
localhost systemd[1]: Starting Mount sysroot...
localhost mount-sysroot[309]: Mounting /sysroot from /dev/vda3 (readwrite)
localhost mount-sysroot[309]: Mount /sysroot success
localhost kernel: TECH PREVIEW: fs-verity on ext4 may not be fully supported. Please review provided documentation for limitations.
localhost kernel: EXT4-fs (vda3): mounted filesystem 76a22bf4-f153-4541-b6c7-0332c0dfaeac r/w with ordered data mode. Quota mode: none.
localhost systemd[1]: Finished Mount sysroot.
```

No systemd-udev in the initramfs

We decided to not use systemd-udev in the initramfs since it is slower than we would have liked. We ended up going with our own minimal approach with just enough functionality to get to the root filesystem with mount-sysroot.

https://gitlab.com/CentOS/automotive/rpms/util-linux-automotive

mount-sysroot.c

No kernel modules in initramfs

- Loading storage kernel modules from the initramfs:
  [0.265061] Run /init as init process
  [0.749703] virtio_blk virtio1: 8/0/0 default/read/poll queues
  [0.753037] virtio_blk virtio1: [vda] 16777216 512-byte logical blocks (8.59 GB/8.00 GiB)
  [0.758552] vda: vda1 vda2 vda3

- Setting your storage drivers to built in gets the storage started sooner.
  [0.202561] virtio_blk virtio1: 8/0/0 default/read/poll queues
  [0.204832] virtio_blk virtio1: [vda] 16777216 512-byte logical blocks (8.59 GB/8.00 GiB)
  [0.207574] vda: vda1 vda2 vda3
  [0.270301] Run /sbin/init as init process

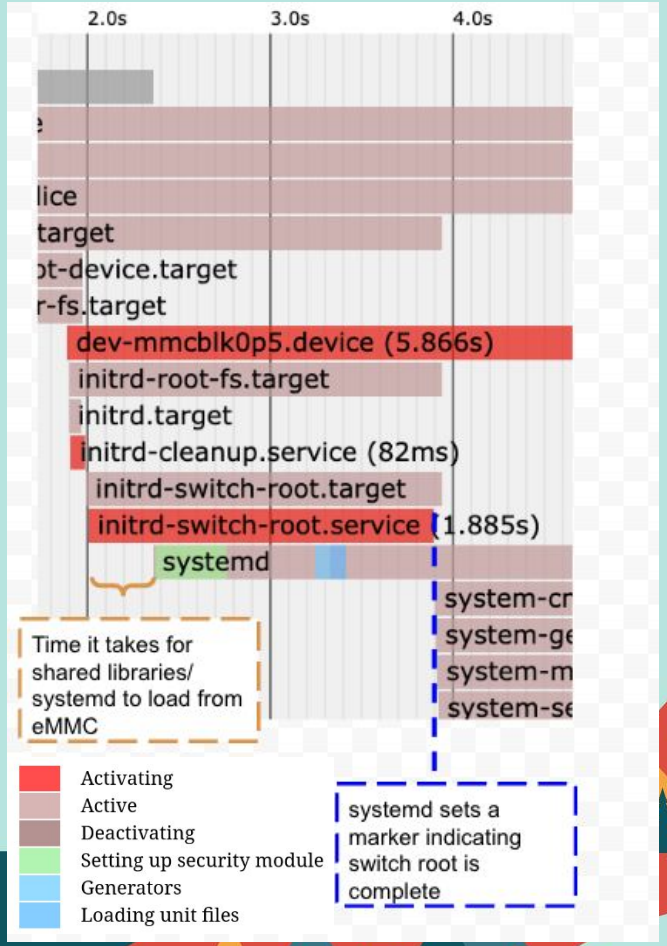
https://gitlab.com/redhat/centos-stream/src/kernel/centos-stream-9/-/merge_requests/3664

initrd-switch-root.service

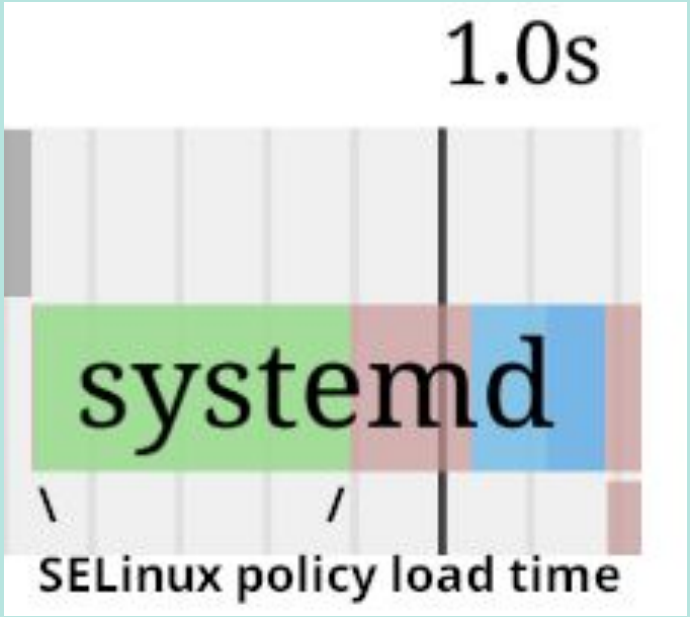
‘systemd-analyze plot’ shows the initrd-switch-root.service can run for a long time, however that’s not entirely accurate.

It finishes once systemd on the root filesystem has fully started:

- Systemd binary / shared libraries from the root
- SELinux policy
- Generators, loading unit files
- On lower performance boards:
  - The rose color after loading the unit files is related to deserializing the systemd state from the initramfs.
  - Additional delay in manager_ready()



Reduce SELinux policy



- Systemd loads the SELinux policy serially on startup.
- Removing unneeded policies can help improve overall boot time.
- Reducing the size of our SELinux policy reduced the policy load time on boot by approximately 50%.

Reduce / Refine udev rules

- Trim down the amount of rules that are defined in udev on the root filesystem (and/or the initramfs). For example, do you need all of the /dev/disk/by-xxx/ symlinks?

- If you run blkid from a udev rule, then see if you can run with '--no-part-details' so that each partition is not examined, and only look at what's stored in the GPT.

- Look at all cases where external programs are called by udev.

udev cold-plug

- Cold-plug: udev replays past events to synchronize devices states.
  - The kernel resends events when udevd kicks in
- This often replays events for low priority devices
  - udev starts early in systemd services bootstrap and a lot of services depends on it.
  - Coldplug can now filter by sub-system
- This can lead to another bottleneck: *SYSTEMD_DEFAULT_MOUNT_RATE_LIMIT_BURST*
  - Stalls mount units on systems with many mounts and few resources (default is 5).

Avoid long-running services from the initramfs

Any processes started from the initramfs need to be short lived, and run quickly. Ideally you should do just enough to get the root filesystem mounted, and start your long-lived processes from the root filesystem.

Why not start long-running services in the initramfs?

- When systemd transitions to the root filesystem, the tmpfs filesystem for the initramfs is deleted. So your running processes whose binaries have been deleted on the filesystem.

- The initrd is a cpio archive, and increasing the size of the initrd is going to increase the kernel boot time since it will need to uncompress and extract the larger cpio archive.

- Any services started from the initramfs will be started before the SELinux policy is loaded, and will run with the *kernel_t* SELinux label.

Why not start long-running services in the initramfs?

- Services started from the initramfs can't depend on almost anything like devices, mounts, services, dbus, etc so it's difficult to develop software of any complexity.
- Adding all of these dependencies to the initramfs will move all of the timing bottlenecks from booting to the root filesystem to the initramfs.
- If you really must start a long-running process from the initramfs, then you should start a second copy from the root and pass the running state. See https://github.com/masneyb/early-service-example for an example.

Early Service Example with socket passing

 # early-service-initrd.service is started inside the initrd.
Apr 05 17:32:16 localhost early-service[323]: Listening on UNIX socket /run/early-service/early-service.sock
Apr 05 17:32:16 localhost early-service[323]: 0
Apr 05 17:32:16 localhost early-service[323]: 1
Apr 05 17:32:16 localhost early-service[323]: 2
Apr 05 17:32:16 localhost early-service[323]: 3
Apr 05 17:32:17 localhost early-service[323]: 4
Apr 05 17:32:17 localhost early-service[323]: 5
Apr 05 17:32:17 localhost early-service[323]: 6
Apr 05 17:32:17 localhost early-service[323]: 7
Apr 05 17:32:17 localhost early-service[323]: 8
Apr 05 17:32:17 localhost early-service[323]: 9
Apr 05 17:32:18 localhost early-service-initrd[323]: ** Message: 17:32:17.649: 10
Apr 05 17:32:18 localhost early-service-initrd[323]: ** Message: 17:32:17.749: 11
Apr 05 17:32:18 localhost early-service-initrd[323]: ** Message: 17:32:17.850: 12
Apr 05 17:32:18 localhost early-service-initrd[323]: ** Message: 17:32:17.950: 13
Apr 05 17:32:18 localhost early-service-initrd[323]: ** Message: 17:32:18.050: 14
Apr 05 17:32:18 localhost early-service-initrd[323]: ** Message: 17:32:18.072: Passing file descriptor for /run/early-service/early-service.sock to other process
Apr 05 17:32:18 localhost early-service-initrd[323]: ** Message: 17:32:18.072: Returning counter to client and terminating the process
Apr 05 17:32:18 localhost early-service[428]: Reading starting position and taking ownership of socket /run/early-service/early-service.sock
Apr 05 17:32:18 localhost early-service[428]: Successfully received file descriptor for /run/early-service/early-service.sock
Apr 05 17:32:18 localhost systemd[1]: early-service-initrd.service: Deactivated successfully.
Apr 05 17:32:18 localhost early-service[428]: 15
Apr 05 17:32:18 localhost early-service[428]: 16
Apr 05 17:32:18 localhost early-service[428]: 17
Apr 05 17:32:18 localhost early-service[428]: 18
 # Only the version started from the root filesystem is now running.

https://github.com/masneyb/early-service-example

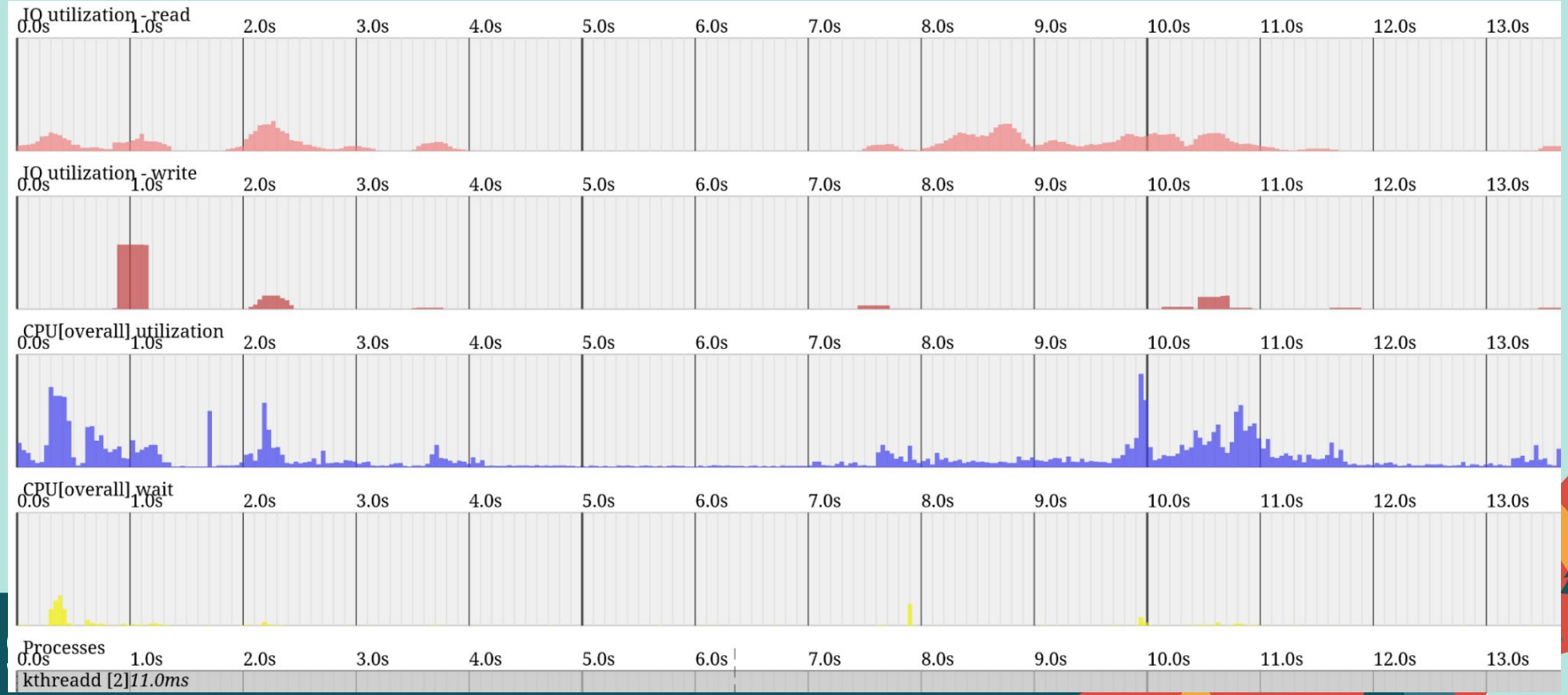
‘systemd-analyze plot’

Included with systemd and shows when each systemd target and service was started, along with how long it took to start Each.

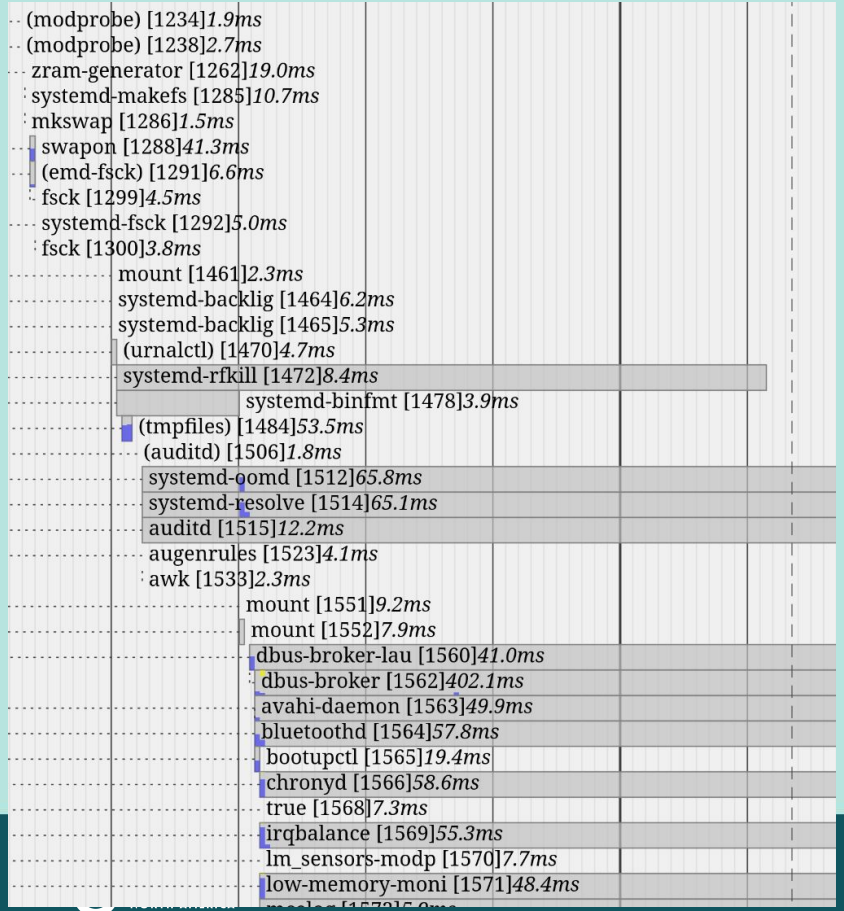
Generally this is always available on distros with systemd.

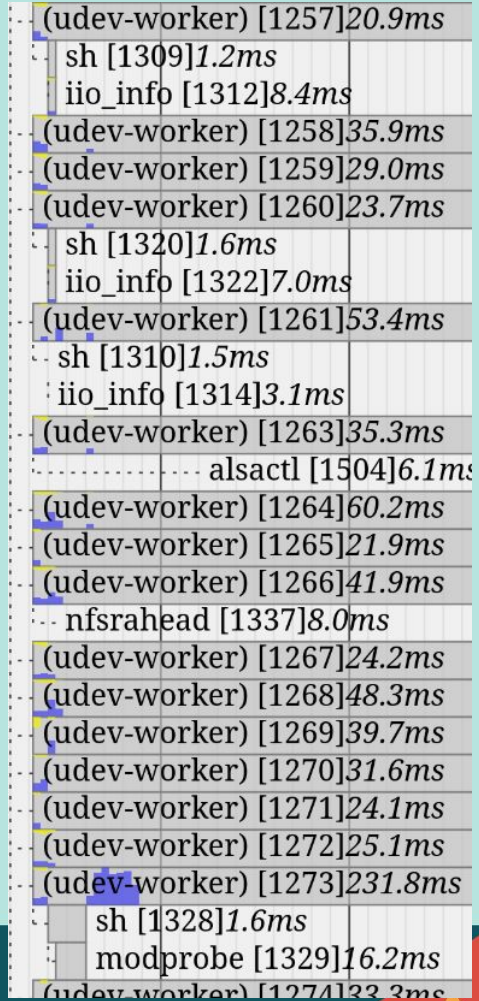
‘systemd-bootchart’ (1 of 2)

Systemd-bootchart generates disk, CPU, and per process statistics on boot.



‘systemd-bootchart plot’ (2)





Ask a LLM: Generate a systemd-bootchart 🤣

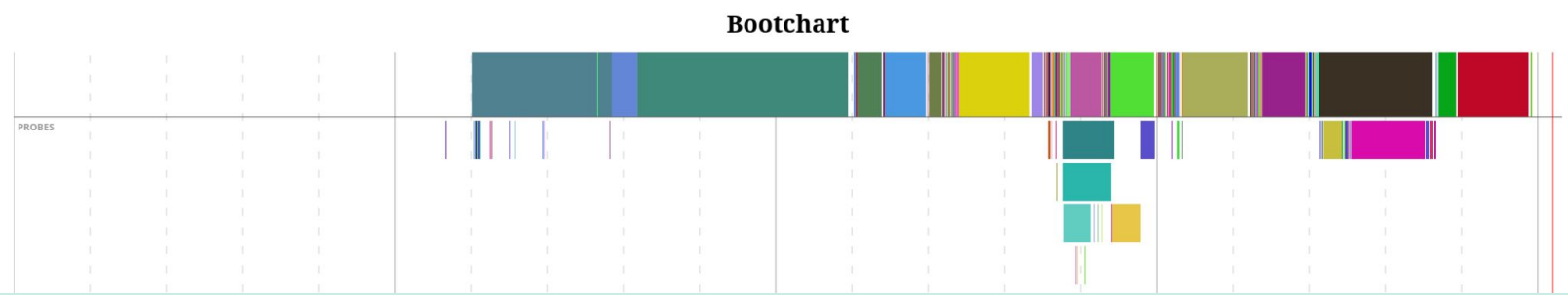
It even generated the MAC window chrome on my Fedora Linux system running GNOME.

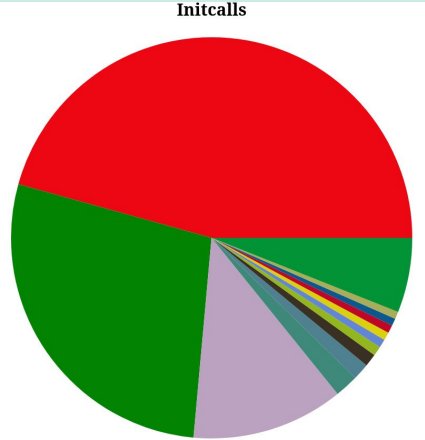
Custom tools

- Boot Time SIG tools, sharing results is greatly appreciated
- Francesco Valla's analyze-initcall-debug.py
  - boot with *initcall_debug quiet*
  - collect the dmesg and process it
- Red Hat's boot-time-analysis
  - Includes cntvct offset (requires kernel patch)

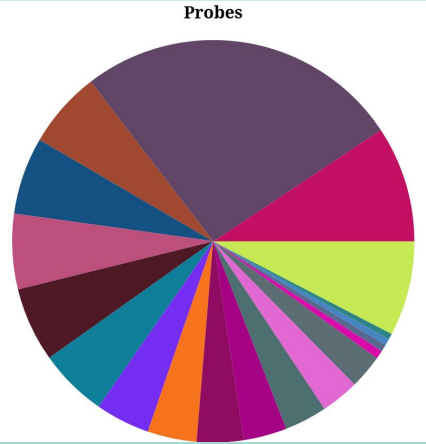
```
Summary:
  810 initcalls have been executed, of which 810 before userspace and 0 after
  0 deferred probes are pending
  23 initcalls/probes failed
  Total boot time: 5172ms
  Init start time: 202ms

---

Top 10 initcall durations:
 * qcom_ethqos_driver_init -> 665459us
 * rtl8152_driver_init -> 405527us
 * dwc3_driver_init -> 178323us
 * init_kprobe_trace -> 27635us
 * of_platform_default_populate_init -> 19531us
 * deferred_probe_initcall -> 14895us
 * pmic_gpio_driver_init -> 11389us
 * net_dev_init -> 9765us
 * jent_mod_init -> 9389us
 * bpf_tcp_ca_kfunc_init -> 9353us

---

Top 10 probe durations:
 * 23040000.ethernet -> 1394754us
 * 1c10000.pci -> 1276358us
 * 1c00000.pci -> 1247229us
 * 3-1:1.0 -> 379771us
 * a600000.usb -> 90096us
 * a800000.usb -> 89882us
 * 0:0:0:2 -> 88400us
 * 0:0:0:4 -> 86857us
 * 0:0:0:6 -> 80502us
 * 3-1 -> 63633us
```

```
[root@localhost ~]# boot_timings
## Timeline of Boot Stage Durations
[=============================================/=/===/=============================================================================================/=>

^ SystemInit: 4483 ms                              ^ Kernel: 194 ms
                                                     ^ InitRD: 393 ms
                                                         ^ Switchroot: 6774 ms

## Durations (ms)
Stage               Components                  Start                 End
SystemInit                                      0.0                   4482.889
                    KernelTimerOffset           0.0                   0.0
                    SystemInitCNTVCT            0.0                   4482.889
[...]
[root@localhost ~]# dmesg | grep cntvct
[    0.000022] time_init: cntvct: epoch=83791813, now=85577291, time=92993 us
[root@localhost ~]# systemctl status -o short-monotonic cntvct@basic.service
[...]
[    5.566986] localhost cntvct@basic[344]: 10.035521s.
[...]
```





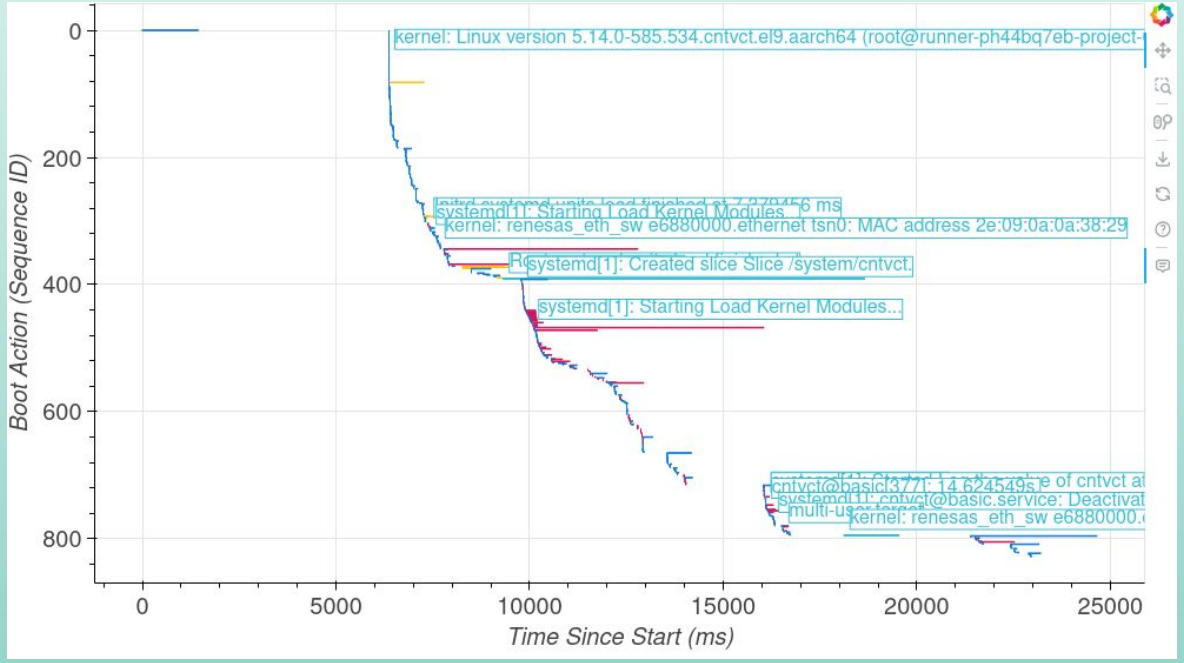
analyze-initcall-debug.py



Name	Status	Duration (us)
qcom_ethqos_driver_init	OK	665459
rtl8152_driver_init	OK	405527
dwc3_driver_init	OK	178323
init_kprobe_trace	OK	27635
of_platform_default_populate_init	OK	19531
deferred_probe_initcall	OK	14895
pmic_gpio_driver_init	OK	11389

Name	Status	Duration (us)
23040000.ethernet	OK	1394754
1c10000.pci	OK	1276358
1c00000.pci	OK	1247229
3-1:1.0	OK	379771
a600000.usb	OK	90096
a800000.usb	OK	89882
0:0:0:2	OK	88400

boot-time-analysis



Q&A

Thanks for listening!

- https://birdcloud.org/boot-time/Front_Page
- https://sigs.centos.org/automotive/

THE LINUX FOUNDATION
OPEN SOURCE SUMMIT
NORTH AMERICA