Linaro

LMG
Mobile

Improving the bootup speed of AOSP

Bernhard “Bero” Rosenkränzer <bero@linaro.org>

ELC 2017-02-23

Quick overview

2 different possible approaches:

- Reduce regular bootup time
  - Problem:
    Lots of initialization needs to be done - stuff needs to start, will always take some time.
    Where can we save time without breaking things?

and

- Investigate suspend-to-disk solutions
  - Problems:
    - No reliable implementation on AOSP so far (being fixed)
    - How to handle power loss without chance to save the current state?
    - Best way to recover from crashes (power off, restore -> restores back to crashed state)
    - Can we just “suspend” a session after first successful boot and “restore” into that ever after (“suspended” again only after updates?) to simulate a fresh boot?

Suspend to disk approaches

- Ported regular Linux suspend scripts
- Essentially working - issues with some non-upstream SoC specific code
- Boot time on armv7hl test hardware (kernel to UI) around 12 seconds (compared to 16 seconds with regular bootup)
- Tux on Ice still being investigated
- Restoring a constant "suspended" image still being done
  - Possible problems with this approach even if (and when) it works:
    - Storage space
    - Potential security issues because restoring would also restore any random generator seeds (can be fixed)
    - No boot animation

Regular bootup: useful tools

- Bootchart: Already built into AOSP

  `touch /data/bootchart/enabled`

  to enable (recorded data can be retrieved with system/core/init/grab-bootchart.sh)
- Systrace
- dmesg, logcat etc.

Reducing regular bootup time: services

- Not all services need to be started before the UI is up...
  - Current patches delay startup of
    - Vibrator service
    - Consumer IR service
    - WiFi NAN and RTT services
    - Network Stats service
    - EthernetService
  - Other services being investigated, including
    - audioflinger
    - adb and other development tools
    - packagemanager

Reducing first bootup time: PackageManager Digest

- Package Manager Scanning
  - Open the AndroidManifest file from APK (using xml parser and zip archive utilities)
  - Read all the relevant attributes starting from version code, version name, permissions etc..
  - check signature
  - Store a package list with all relevant information in `/data/system/packages.list` and `/data/system/packages.xml`
- On second boot
  - Check if package has been updated, if yes then entire package info will be re-scanned
  - If not, existing information will be used from packages.xml and packages.list
- Possible "fix":
  - Include packages.list and package.xml with information about preinstalled apps in userdata.img
  - On first boot, PackageManager use existing digest and save up time for individual package scanning.
  - Total sec saved on first boot is approx ~4.x sec (but almost no time saved on subsequent boots)

Reducing regular bootup time: init

- Init and init.rc were already optimized quite well, but:
  - Frequent construct in initrc files:
    mkdir x 0700 user group
    mkdir y 0700 user group
    Or
    chown user group x
    chmod 0755 x
    chown user group y
    chmod 0775 y
  - There may be room for saving some parsing time by extending the syntax of mkdir and turning those all into one line, e.g.
    “chmod_chown x 0666 user group” OR “mkdir_parallel x,0770,user1,group1 y,0700,user2,group2”
- Doesn’t have the same effect this would have on a traditional shell based init because mkdir/chown/... are implemented internally, so the overhead of launching a new process for each call isn’t there in the first place.

Reducing regular bootup time: preloading

- AOSP preloads classes to speed up application startup...
  - But that slows down system startup.
    We can delay preloading relevant big classes until after the UI is up:
    - WebView factory
    - Anything listed in /system/etc/preloaded-classes -- among others
      - Most android.* classes
      - Most of the Java core library:
        - java.io.*
        - java.lang.*
        - java.math.*
        - java.net.*
        - java.nio.*
        - java.security.*
        - java.text.*
        - java.util.*
        - javax.net.ssl.*
        - javax.security.*
        - sun.security.*
      - libcore.*

Reducing regular bootup time: preloading

- Current patches delay all class preloading
  - This may not be the smartest thing to do since even the window manager and launcher -- needed for successful bootup -- will need those classes.
  - It will probably be more efficient to split classes into
    - /system/etc/preloaded-classes
      Same purpose as now: Anything that should be preloaded for applications, to be loaded after the system is up
    - /system/etc/early-preloaded-classes
      Classes that should be preloaded early (at the point where upstream AOSP preloads classes) because even the launcher or other bits needed to complete "boot to UI" phase make use of them
    - This should be left in configurable list files instead of hardcoding, since AOSP forks with different launchers (e.g. specialized STB or automotive launcher) may require different classes at an earlier time.

Reducing regular bootup time: PM

- Force highest CPU frequency during boot, pin background tasks to LITTLE CPUs later
    - Power management can get in the way of fast bootup - setting up parameters designed to save power (pinning background tasks to LITTLE CPUs in ASMP environments etc.) at a later point can help.

Reducing regular bootup time: I/O

- Bootup is I/O heavy…
  - Identify the best I/O scheduler for bootup
  - Optimize read-ahead settings
  - Best settings need to be identified for every possible target board - even with the 3 boards we’ve been targeting, results didn’t match

Reducing regular bootup time: Kernel features

- Filesystem
  - Squashfs tends to result in better bootup time than ext4
- Kernel decompression
  - Switching kernel compression from zlib to lz4 increases bootup speed, but adds around 2.6 MB memory requirement (on armv7hl sample hardware)
  - Zstd decompression currently being ported to the kernel - it’s known to be both fast and efficient, and might provide the best of both worlds.

Reducing regular bootup time: Kernel modules

- Modular kernel
  - Building some kernel components as modules and only loading them on demand should help speed up booting some more.
  - Currently AOSP doesn't use kernel modules at all, but this is changing in AOSP master. `modprobe` is enabled in AOSP master's toybox build.
  - I/O is slow -- even ripping modules that don't have any code run at initialization time out will speed up booting

Reducing regular bootup time: libc

- Make sure we use the fastest memcpy/memcmp/... functions available
  - Memcpy and friends are used heavily during bootup
  - We’ve made sure Bionic has the fastest implementations available before - but since then, newlib, musl etc. have made progress. Need to make sure we still have the fastest implementations there both for bootup performance and runtime performance
- Also, use current compilers to generate the best possible code

Questions?

Bernhard “Bero” Rosenkränzer <bero@linaro.org>

2017-01-09