Google

Chrome OS

The kernel in the hands of millions of users

September 2021

Alex Levin - levinale@google.com
Jesse Barnes - jsbarnes@google.com
Feel free to send CVs and questions

Agenda

- ChromeOS kernel lingo -
  - Rebase - forward port CHROMIUM patches to current tree
  - Uprev - debug & deploy new kernel to devices
  - Continuous rebase - keep CHROMIUM patches fresh against latest -rc, get test results
- ChromeOS Test coverage for upstream rc kernel
- Upstream agony
- Upstream first
- Partners (vendors, O[E|D]Ms and SoC manufacturers) and upstream

Kernel in Chrome OS

- Active CROS kernel versions
  - Multiple active (shipping) kernel version:

```
~/chromiumos/src/third_party/kernel/➜  kernel ls -la
drwxr-x---  26 levinale primarygroup 4096 Aug  2 11:45 upstream
drwxr-x---  26 levinale primarygroup 4096 Jun  1 15:14 v3.18
drwxr-x---  26 levinale primarygroup 4096 Aug  2 11:45 v4.14
drwxr-x---  26 levinale primarygroup 4096 Jun  1 15:14 v4.14-gw
drwxr-x---  27 levinale primarygroup 4096 Aug  2 11:45 v4.19
drwxr-x---  27 levinale primarygroup 4096 Jun  1 15:14 v4.19-ht
drwxr-x---  27 levinale primarygroup 4096 Jun  1 15:14 v4.19-manatee
drwxr-x---  27 levinale primarygroup 4096 Aug  2 11:45 v4.4
drwxr-x---  26 levinale primarygroup 4096 Aug  2 11:45 v5.10
drwxr-x---  26 levinale primarygroup 4096 Aug  2 11:45 v5.10-arcvm
drwxr-x---  26 levinale primarygroup 4096 Aug  2 11:45 v5.4
drwxr-x---  26 levinale primarygroup 4096 Aug  2 11:45 v5.4-arcvm
drwxr-x---  26 levinale primarygroup 4096 Jun  1 15:15 v5.4-manatee
```

- Each of these kernel versions map to multiple platforms shipping with it
- The kernel version for a platform is selected at birth (or bringup).

Kernel Rebase

Once in a Blue moon (or every LTS release) we rebase to a new kernel (5.15 will start soon).

The need to rebase comes from:

- New platforms are being developed
  - Easier to cherrypick (sometimes hundreds of patches at a time) on top of the newer kernels
- Our desire to keep as close as possible to upstream

A rebase (or at least it used to be) is a process that involves multiple teams - splitting the kernel into topic branches and each team resolves/debuggs its own topic branch.

Kernel Uprev

- Moving a platform from kernelX to KernelY is called an uprev.
- A test driven activity mostly.
  - Need to pass ChromeOS tests (and CTS).
- Have to deal with some upstream bugs & regressions
  - Often due to changes during upstreaming of vendor code
  - Trying to improve this with kernelci.org (seeding with lots of Chromebooks)
  - Still need better test coverage, both internally and externally
  - FDO Graphics CI serves as a good model here
- Most time is spent looking for problems relative to the old kernel
  - Fixing failing tests.
  - Digging through feedback reports from users, trying to figure out of bugs are regressions
- Non-upstream stuff causes the most pain (surprise!)
  - Graphics drivers, some pre-SoF sound stuff, etc.

Goal is update every device every other year with a new kernel. Two live versions in the field, one in development.

Uprevs are unpredictable

- Upreving a platform from 4.19 to 5.4 is fun! (isn't it?)
  - In most cases the platform has breakage upstream in most components (e.g. audio, i2c, performance, etc.).
- Hard to plan for - the depth of the rabbit hole is unclear before you dive.
  - Becomes a resourcing/scheduling burden
- A lengthy uprev consumes a lot of lab equipment (2x on the testing capacity).

Continuous rebase and continuous testing

- To map the unexpected, rebase on top of every single RC
- Test every single RC to track for regressions
- Report breakage to ChromeOS teams
- Report regressions upstream(still in the works)
- Send patches upstream (e.g. 1,2,3)

Report the failures (ideally automatic)

Internally we have scaled quite well - teams are looking at all the failures (hundreds of bugs opened and resolved)

The upstream story needs more work - we are starting to explore how to best integrate with upstream. Investing money & time in KernelCI as part of this.

ChromeOS Upstream First (link)

- Upstream first: We aim to get all patches accepted upstream
  - Upstreaming means sending patch to some mailing list, getting it reviewed there
  - Maintainer then picks up the patch, puts it in a git tree, and later asks Linus to merge in main tree.
  - Most common types of patches:
    - **UPSTREAM**: The commit was accepted upstream, and is available in a later kernel version.
      - Must contain (cherry picked from commit 7c761b593e2c1dc6bc6c0c15ec338af1f00cabd7)
      - We must have reasonable confidence that the commit ID won't change (if in Linus tree, surely, otherwise, it depends). If unsure, use **FROMGIT** tag instead.
      - Patch must apply cleanly, otherwise mark as **BACKPORT**, and indicate what changed.
    - **FROMLIST**: The CL was posted upstream, and likely not in its final version.
      - Must contain (*am from https://patchwork.kernel.org/patch/9768741/*)
      - Do this when in a rush (we like boards to boot, bugs to be fixed). We can always revert the patch and pick up a UPSTREAM later.
      - Sometimes used for patches that have no chance of being accepted upstream in their current form (e.g. maintainer asks for refactoring, etc.)
    - **CHROMIUM**: CL that have zero chances of being accepted upstream
      - Chrome OS config options (more about that later)
      - Graphics drivers for ARM (upstream does not like it when the userspace driver is closed source)
      - Experiments for data gathering (e.g. early versions of MGLRU, core scheduling)

Upstreaming agony

We *want* to upstream everything. It makes Linux better and our lives easier. *However:*

- High variability in maintainer responsiveness
  - Some subsystems are really great
  - Some armitecture maintainers are not as easy to work with
  - Some subsystems are just stuck (e.g. memory management)
- Replies often come with “helpful” suggestions of radical product redesign
  - E.g. preempt count passthrough for VMs to improve scheduling of guests
- Plus usual stuff, e.g. “oh sure we can apply this two liner… *after* you rewrite the subsystem”

Wishlist:

- Consistent maintainer responsiveness and acceptance criteria
  - A maintainer CoC or expectations doc?
- More data driven decision making (e.g. which benchmarks are generally agreed to be important for each subsystem)
- More openness to experimentation
  - How can we enable this?

Partners and upstream

- Generally - no CHROMIUM is allowed
- But some cases are approved
  - As a temporary workaround until the upstream story is well digested
- Actually landing FROM[GIT|LIST] upstream
- Reverting temporary solutions and replacing them with upstream patches
  - Tracked in bugs assigned to partners.


ANY
QUESTIONS
?

Informational slides

Build & flash cros-kernel (link) (from our sdk)

- Build a kernel (e.g. for caroline)
  - setup_board --board=caroline
  - cros_workon-start --board=caroline chromeos-kernel-4_19
  - emerge-caroline chromeos-kernel-4_19
- In case you want to build a whole chromium image:
  - USE="pcserial tty_console_ttyS1" ./build_packages --board=caroline
  - ./build_image --enable_serial='ttyS1,115200n8' --board=caroline --noenable_rootfs_verification test
- Update the board with your custom kernel
  - ./update_kernel.sh --remote <IP_ADDR>
- Flash the image to a USB
  - cros flash usb:// ../build/images/caroline/<latest>/chromiumos_test_image.bin
  - Make sure to enable crossystem dev_boot_legacy=1 to boot from usb (crtl + U)

Kernel in Chrome OS (link)

- Must flash test image for ssh and other (most) useful debug tools
  - To make the roofs writable:
    - /usr/share/vboot/bin/make_dev_ssd.sh --remove_rootfs_verification --force

Kernel in Chrome OS (link)

- Switch to terminal (tty)
  - Once developer mode is enabled
    - Esc+F3 (refresh)+power - takes to recovery screen
    - Ctrl + D
  - Ctrl+Alt+F2 (forward arrow)
  - Test image root default password is “test0000”

Debugging in case stuff doesn't work(great! link)

- Enable serial console
  - USE="pcserial **tty_console_ttyS0**" ./build_packages --board=caroline
  - ./build_image **--enable_serial='ttyS0,115200n8'** --board=caroline --noenable_rootfs_verification test

- Debugging using prints
  - Add printks in strategic places (dev_[info/warn/err] or pr_[info/warn/err])
    - pr_<level>: Slightly shorter than printk(KERN_<LEVEL>)
    - dev_<level>: Standardized device information: dev_driver_string, then dev_name
    - dev_dbg/pr_dbg in the kernel code can be enabled by setting #define DEBUG at the top of the source file (before all includes).
  - Adding dump_stack calls in places may also be very useful
  - BUG/WARN provide nice backtraces.

Debugging in case stuff doesn't work(great! link)

- kasan (**K**ernel **A**ddress **san**itizer):
  - Compile the kernel using USE=ubsan and USE=kasan
  - Kasan is a dynamic memory error detector. It provides a fast and comprehensive solution for finding use-after-free and out-of-bounds bugs.
    - Uses compiler instrumentation for checking every memory access - expect to pay performance
  - kasan prints a report in case of a bug found
    - The header of the report describes what kind of bug happened and what kind of access caused it.
    - In the last section the report shows memory state around the accessed address. For better understanding - read the link.
- ubsan (Undefined Behavior Sanitizer):
  - UBSAN uses compile-time instrumentation to catch undefined behavior (UB).
  - The compiler inserts code that perform certain kinds of checks before operations that may cause UB. If check fails (i.e. UB detected) __ubsan_handle_* function called to print error message.
  - Produces a report with the file/line that caused UB.
  - Allows to sanitize per file/directory (limit the performance cost).

Debugging in case stuff doesn't work(great! link)

- kmemleak
  - Kmemleak provides a way of detecting possible kernel memory leaks
  - A similar method is used by the Valgrind tool (memcheck --leak-check) to detect the memory leaks in user-space
  - A kernel thread scans the memory every 10 minutes (by default) and prints the number of new unreferenced objects found.
- Testing your code for failure (failslub)
  - The kernel has a debugfs API to Configure fault-injection capabilities behavior
  - This helps test code when failure happens
  - Allows to introduce new failures
- In case of an oops the chromebook will reboot but the logs of the oops can be obtained
  - cat /dev/pstore/console_rampoops