Shared Logging with the Linux Kernel

mentor embedded

mentor.com/embedded

Android is a trademark of Google Inc. Use of this trademark is subject to Google Permissions.
Linux is the registered trademark of Linus Torvalds in the U.S. and other countries.
Qt is a registered trade mark of Digia Plc and/or its subsidiaries. All other trademarks mentioned in this document are trademarks of their respective owners.

Outline

- What and why of shared logging?
- Hey! Haven't I seen this before?
- Kernel logging structures, then and now
- Design and Implementation
- Live Demo
- Current status
- Q&A / Discussion

What is shared logging?

- It's really already in the name, but I'll spell it out below
  - (no, not **#exactsteps** ☺)
- Simply put, the bootloader and the kernel can read and write log entries for themselves normally

  and read log entries from the other

- For the bootloader, this implies that log entries persist past reboots. For now, I have focused on shared volatile RAM, but this could work for NV storage of logs as well.

Why would we want shared logging?

- Imagine debugging without logging.
  - ☺
- Most common use case:
  - Post-mortem analysis of a failed bootloader boot
  - Post-mortem analysis of a failed kernel boot
- Other useful cases:
  - Performance tweaking
  - Boot timing analysis
  - Boot sequencing analysis
  - Boot and system debugging
- Shared logging provides you with another tool in the box to use when you need it

Haven’t we seen this before?

- Yes!
- From git history, back in late 2002, Klaus Heydeck added support for a shared memory buffer that could be passed to the kernel to be used for shared logging.
- AFAICT, this feature was only supported in the Denx’s kernels and not for all architectures. (PPC only?)
- Focus seems to have been primarily on being able to see bootloader entries in the kernel
- Does not appear to have been widely used
- Unfortunately, the feature has suffered bit rot over time and changes in the kernel logging structures broke it (more on those changes later)

Kernel logging structures (then)

- For a considerable time prior to 3.4, the kernel log was a byte-indexed array of characters
- Structure and implementation contained in printk.c
- Buffer space was declared as a static global inside printk.c
- Indices provided for logging start, logging end, and console start locations in the buffer
- Simple implementation
- Fairly easy to support by the bootloader

Kernel logging structures (now)

- In May 2012, Kay Sievers' patch changed the structure to a variable length record with a fixed header
- Structure and implementation still contained in printk.c
- Buffer space still declared as a static global inside printk.c
- The header is 16 bytes and includes the timestamp in binary form
- More complex. Has more pointers for tracking
  - Sequence and index for: first, next, clear, & syslog

A few observations

- The shift to a record based structure in the kernel introduced more pointers to manage for the handoff between the bootloader and the kernel to occur correctly
- Global static declarations in the kernel makes the logging structures available as soon as the C runtime is available (important later)
- Using global statics structures complicates sharing the log entries

Some goals for reviving this capability

- Available all the time
  - Must have negligible impact on regular boots
- Portable across bootloaders
  - uBoot would provide POC reference, but should be easy to port
- Support arbitrary location for logging buffer
  - Allows the bootloader to specify an arbitrary location to the kernel
- Minimize 'lost' memory due to global static allocations
- Provide self-checking that ensured correct operation in the face of incompatible entries seen by the bootloader of the kernel
- Provide as an 'opt-in' for both bootloader and kernel
- NOTE: the focus was on getting a bootloader to write a format that the kernel understands, not to provide a new, general mechanism for sharing

Interface design

- To address the number of parameters needed to be passed into the kernel, I added a control block structure
- The control block encapsulates all of the necessary logging information including structure size, various indices, and buffer locations for sharing purposes
- Allows a single pointer location for the control block to change where the log information is being written
- Allows the bootloader to pass a single parameter to the kernel
- In theory, allows the kernel to adopt the CB and start writing immediately to the next location in the buffer (O(1) operation)
  - In practice, there are wrinkles

How to pass the CB to the kernel?

- Fixed, well known location
  - Used by the original shared log feature
  - Works, but is very brittle
    - Relies on a calculation of the end of RAM to align between the kernel and the bootloader
    - Doesn't always work!
- Command line
  - Initial approach used to revitalize the feature
  - Very flexible and allows for dynamic setting by the user
  - There's a small performance hit that occurs during log coalescing
    - This is O(n) based on the number of bootloader log entries and kernel entries written when the coalescing occurs (more later)
  - Personally, I greatly prefer this approach
  - Acceptable upstream?

How to pass the CB to the kernel? (2)

- DeviceTree
  - Second approach used to revitalize feature
  - Fixed at DT compile time
  - Again, there is a small performance hit that occurs during log coalescing, albeit slightly reduced from before
    - This is O(n) based on the number of bootloader log entries and kernel entries written when the coalescing occurs (more later)
  - Perhaps more acceptable upstream?

Bootloader implementation

- Tested with a Boundary Devices Sabre-Lite (i.MX6Q)
- Built against a 2014.7 boundary devices u-boot
- Existing log entry format in uBoot was very different from that in the kernel
- However, uBoot already had the concept of a versioned log format
- So, introduced a new log format (v3) to be compatible with the kernel format
- Log version is controlled by an environment variable, so user can dynamically 'opt-in', as desired, using standard setenv commands
- The log CB and the log size are also controlled via environment variables

Kernel implementation

- Based on the FSL vendor kernel, v3.10
- Relocated all the sequence and indices to a CB
- Added support for re-pointing the CB from a global static to one passed in to the kernel
- Initially, used command line arguments to pass the necessary pointer to the CB
- During command line processing, the values for the shared log are parsed and captured for later use
- After mm_init(), the function setup_ext_logbuff() gets called, which halts the logging temporarily and coalesces the entries together
  - This can create a small time hit as the entries are copied from the previously used buffer for the kernel into the bootloader provided buffer
  - This is O(n) because it depends on the number of entries from the bootloader and the number of entries from the kernel when this is run
  - Luckily, neither is very large, but it would be nice to do away with that hit entirely

Kernel implementation (2)

- Switching to using the DT worked, but didn't help very much
- By default, DT processing still occurs after logging events have already started to be delivered into the kernel log
- Same need to coalesce entries together occurs
- The DT processing did occur earlier, so, it meant fewer entries to coalesce
  - Still not where we want to be
- Accessing the raw DT data earlier in init was possible, but I did not get the buffer mapped into memory properly before having to put this work aside for other priorities
  - ☹
  - In theory, should be workable, but needs to be proven out

Some gotchas

- Physical vs virtual addressing
  - Bootloader uses physical
  - Kernel uses both, depending on where you are in the code
  - Making sure the right addresses are used is critical
- Mapped memory vs unmapped memory
  - Kernel memory gets mapped in stages
  - Make sure that the memory you are attempting to address is mapped in before you use it

SOURCE CODE

DEMO

Current Status

- This works internally against the older kernel and uBoot
- Unfortunately, I have had to work on other tasks for the last few months instead of this feature
- So, these changes haven't been cleaned up or ported forward for upstream submission, **yet**.
- Also, I still want to remove the small hit caused by the coalescing process
  - Need to initialize the buffer early enough to make coalescing un-necessary
- I plan to tackle these issues when I return home and expect I will have something submitted upstream soon
  - In the interest of getting this out there, I am leaning towards submitting the command line version as a working POC and follow up with improvements later

Q&A DISCUSSION