Shared Logging with the Linux Kernel

!!Part Deux!!

Sean Hudson
Embedded Linux Architect

mentor.com/embedded
Who am I?

I am an embedded Linux architect and Member of Technical Staff at Mentor Graphics. I have worked on embedded devices since 1996. I started working with Linux as a hobbyist in 1999 and professionally with embedded Linux in 2006. In OSS, I have been involved with the Yocto Project since it's public announcement in 2010, have served on the YP Advisory Board for two different companies, and am currently a member of the OpenEmbedded Board.
Why “Part Deux”? 

- To provide an update to my talk at ELCE 2015 in Dublin 
  - Slides for previous presentation here: 
  - Video of previous presentation here: 
    - https://www.youtube.com/watch?v=E4h1Of8zyVg 

- Because I get to make a silly cultural reference
Outline

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

- Simply put, both the bootloader and the kernel can:
  - read and write log entries for themselves normally and
  - read log entries from the other
  - read multiple boot cycles

- The bootloader can also:
  - Dynamically specify a shared memory location to use for logging

- In order for the bootloader to read kernel entries and to allow multiple boot cycles, log entries must persist past reboots. For now, I have focused on shared volatile RAM, but this might work for NV storage of logs as well, ala pstore.
Why would we want shared logging?

- Imagine debugging without logging.
  - 😊

- Most common use case:
  - Post-mortem analysis of a failed boot

- Other useful cases:
  - Performance tweaking
  - Boot timing analysis
  - Boot sequencing analysis
  - Boot and system debugging

- Not a silver bullet!
  - 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 suffered bit rot over time and changes in the kernel logging structures broke it (more on those changes later)
What about pstore and ramoops

- This question came up in Dublin
- From a quick review, they appear to serve slightly different purposes
- They both rely on small, pre-allocated regions of memory
- Perhaps these could be integrated in some fashion
- Certainly, this is an area for future exploration
- Anyone know of additional features that I should look at?

References:
- https://www.kernel.org/doc/Documentation/ABI/testing/pstore
Kernel logging structures (then)

- As far back as 2.6.11, the first git commit in my tree, the kernel log was a byte-indexed array of characters with a simple 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 (then)

```c
/*
 * logbuf_lock protects log_buf, log_start, log_end, con_start and logged_chars.
 * It is also used in interesting ways to provide interlocking in
 * release_console_sem().
 */
static DEFINE_SPINLOCK(logbuf_lock);

static char __log_buf[__LOG_BUF_LEN];
static char *log_buf = __log_buf;
static int log_buf_len = __LOG_BUF_LEN;

#define LOG_BUF_MASK (log_buf_len-1)
#define LOG_BUF(idx) (log_buf[(idx) & LOG_BUF_MASK])

/*
 * The indices into log_buf are not constrained to log_buf_len - they
 * must be masked before subscripting
 */
static unsigned long log_start; /* Index into log_buf: next char to be read by
static unsigned long con_start; /* Index into log_buf: next char to be sent to
static unsigned long log_end;  /* Index into log_buf: most-recently-written-
static unsigned long logged_chars; /* Number of chars produced since last read
```
Kernel logging structures (post 2012)

- 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 fixed and includes the timestamp
- More complex. Has more pointers for tracking
  - Sequence and index for: first, next, clear, & syslog
Kernel logging structures (post 2012)

```c
enum log_flags {
    LOG_NOCONS = 1,   /* already flushed, do not print to console */
    LOG.NewLine = 2, /* text ended with a newline */
    LOG_PREFIX = 4,   /* text started with a prefix */
    LOG_CONT = 8,     /* text is a fragment of a continuation line */
};

struct printk_log {
    u64 ts_nsec;       /* timestamp in nanoseconds */
    u16 len;           /* length of entire record */
    u16 text_len;      /* length of text buffer */
    u16 dict_len;      /* length of dictionary buffer */
    u8 facility;       /* syslog facility */
    u8 flags;          /* internal record flags */
    u8 level;          /* syslog level */
}

#ifdef CONFIG_HAVE_EFFICIENT_UNALIGNED_ACCESS
__packed __aligned(4)
#endif

/*
 * The logbuf_lock protects kmsg buffer, indices, counters. This can be taken
 * within the scheduler's rq lock. It must be released before calling
 * console_unlock() or anything else that might wake up a process.
 */
```

[0] 0:vim*
Kernel logging structures (post 2012)

```c
#ifdef CONFIG_PRINTK
DECLARE_WAIT_QUEUE_HEAD(log_wait);
/* the next printk record to read by syslog(READ) or /proc/kmsg */
static u64 syslog_seq;
static u32 syslog_idx;
static enum log_flags syslog_prev;
static size_t syslog_partial;

/* index and sequence number of the first record stored in the buffer */
static u64 log_first_seq;
static u32 log_first_idx;

/* index and sequence number of the next record to store in the buffer */
static u64 log_next_seq;
static u32 log_next_idx;

/* the next printk record to write to the console */
static u64 console_seq;
static u32 console_idx;
static enum log_flags console_prev;

/* the next printk record to read after the last 'clear' command */
static u64 clear_seq;
static u32 clear_idx;
```

[0] 0:vim*
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.
Revised goals (since last time)

- The original focus for this feature was on getting a bootloader to write a format that the kernel understood, not to provide a new, general mechanism for sharing.
- My goals are slightly different.
- Available all the time
  - Must have negligible or no impact on regular boots
- Portable across bootloaders and architectures
  - uBoot would provide POC reference, but should be easy to port
- Support dynamic, 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
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.
Kernel logging structures (proposed)

* The optional key/value pairs are attached as continuation lines starting
* with a space character and terminated by a newline. All possible
* non-printable characters are escaped in the "\xff" notation.
*/

enum log_flags {
    LOG_NOCONS = 1, /* already flushed, do not print to console */
    LOG_NEWLINE = 2, /* text ended with a newline */
    LOG_PREFIX = 4, /* text started with a prefix */
    LOG_CONT = 8, /* text is a fragment of a continuation line */
};

struct printk_log {
    u32 log_magic;       /* sanity check number */
    u16 len;             /* length of entire record */
    u16 text_len;        /* length of text buffer */
    u16 dict_len;        /* length of dictionary buffer */
    u8 facility;         /* syslog facility */
    u8 flags;            /* internal record flags */
    u8 level;            /* syslog level */
    u64 ts_nsec;         /* timestamp in nanoseconds */
};
Kernel logging structures (proposed)
How to pass the CB to the kernel?

- Fixed, well known location
  - Used by the original shared log feature
  - Used to work, but is brittle/broken
    - 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
  - 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
  - Personally, I greatly prefer this approach
  - Acceptable upstream?
How to pass the CB to the kernel? (2)

- DeviceTree
  - Second approach
  - Fixed at DT compile time
  - Used OF functions to extract information from DT
    - Personally found this a bit difficult to work with
  - Log coalescing still occurred, 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
  - Perhaps more acceptable upstream?
How to pass the CB to the kernel? (3)

- DT + command line arg
  - Third approach
  - Using reserved memory areas in the DT relies on existing infrastructure and ‘just works’
    - Avoids platform specific code for memory reservation too
    - In the UBoot POC, this utilizes the mainline fdt features to modify the DT in a live manner
    - This puts the responsibility on the bootloader to ensure memory is reserved
  - Uses command line parameter to specify memory location of lcb
  - Log coalescing still occurs
Bootloader POC implementation

- 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
- I dropped much of the uBoot env control variables to simplify the design and due to issues encountered during testing
Bootloader upstream status

- Ported to the mainline internally
- Some additional cleanup/refactoring is still needed
- Patches are not submitted upstream as of yet. 😞
Kernel implementation

- 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
- Uses command line to pass the necessary pointer to the lcb
- 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
Kernel upstream status

- Refactoring the code since last time dropped all arch specific code
- Almost all changes are located in printk.h/printk.c
  - Exceptions are: Kconfig and main.c
- Ported to the mainline kernel as of 4.8rc
- Patches submitted to LKML on 2016/09/29
- V2 submitted to LKML on 2016/10/04
- Also available on github here: https://github.com/darknighte/linux/tree/for_review_v2
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

- **Structure packing**
  - Packed structures are bad for portability
  - Had to manually re-order the header struct to make it align

- Also, mucking around in init/early init is fraught with peril and quiet failures.
Some gotchas (2)

- Porting to mainline
  - Patches ported pretty easily and compiled pretty easily
  - Reserved memory regions changed

- Building
  - Building uboot for x86 has been non-trivial
  - Creating test builds with same toolchain

- Testing
  - Initial patch submission to the kernel got a failure for kernel-ci in about 10 mins. 😞
  - Turns out that turning off CONFIG_LOGBUFFER was fine, but turning off CONFIG_PRINTK wasn’t.
Planned and possible future work

- Complete cleanup of U-Boot patches and submit
- Build U-Boot for x86 POC
- Investigate OF extraction of lcb pointer during early boot to remove static global buffer in printk.c
- Investigate timer handoff to kernel for single time base
- Perhaps augment U-Boot env settings to dynamically shift the buffer location and relocate entries
- Investigate coreboot and implement similar feature
Q&A DISCUSSION