printk: Why is it so complicated?

<john.ogness@linutronix.de>

Linux Plumbers Conference 2019

Lisbon, Portugal

Disclaimer

As of 2019-09-09, none of the rework code presented here has been accepted for mainline Linux.

Requirements

Why is `printk()` so complicated?

Basic Requirements

- called from **any** context
  (including scheduler and NMI)
- stores messages into a ringbuffer
  (made available to users via syslog, `/dev/kmsg`, kmsg-dump)
- pushes messages out to console(s)

Advanced Requirements

- ringbuffer should not be missing any messages upon crash/hang
  (requires synchronization with any other context, including NMI)
- consoles should not have missed any messages upon crash/hang
  (calls into console drivers that do ... *stuff*)
- should not interfere with normal operation
  (example: inserting a USB stick should not cause latency spikes)

Requirements

Why is `printk()` so complicated?

"If it is part of printk, it is already implicitly on every line of code." [1]

[1] John Ogness, https://lkml.kernel.org/r/8736nyov8f.fsf@linutronix.de

A Brief History of printk

Linux 0.01 (1991-09) `kernel/printk.c`

```
static char buf[1024];

int printk(const char *fmt, ...)
{
        va_list args;
        int i;

        va_start(args, fmt);
        i=vsprintf(buf,fmt,args);
        va_end(args);
        __asm__("push %%fs\n\t"
                "push %%ds\n\t"
                "pop %%fs\n\t"
                "pushl %0\n\t"
                "pushl $_buf\n\t"
                "pushl $0\n\t"
                "call _tty_write\n\t"
                "addl $8,%%esp\n\t"
                "popl %0\n\t"
                "pop %%fs"
                ::"r" (i):"ax","cx","dx");
        return i;
}
```

A Brief History of printk

0.01/1991-09: **direct synchronous printing to terminal**

0.96a/1992-05: **ringbuffer** (4K), **syslog**, variable "log_wait"

0.99.7A/1993-03: variable "log_buf" (4K), console registration, **upon registration console prints existing ringbuffer**

0.99.13k/1993-09: **loglevels** (encoded as "<level>" in messages)

0.99.14/1993-11: interrupts disabled for ringbuffer store and console printing

2.1.31/1997-03: multiple console support, console write() callback, **each message printed synchronously to all registered consoles**

2.1.80/1998-01: spinlock "console_lock", ringbuffer store and console printing under spinlock

2.4.0/2000-10: bust_spinlocks(), **re-init console_lock on crash/lockup**

2.4.10/2001-08: variable "console_sem" (protects console driver system) replaces "console_lock", variable "logbuf_lock" (protects ringbuffer), variable "oops_in_progress" (re-inits logbuf_lock and console_sem), call_console_drivers() function, called with: "logbuf_lock unlocked, irqs enabled, console_sem held", call_console_drivers() called from release_console_sem(), returns if console_sem held (**printk now non-synchronous**)

A Brief History of printk

2.5.51/2002-12: /dev/kmsg to printk from userspace

2.5.53/2002-12: do not console print if printk-CPU is not online

2.6.0/2003-10: variable "log_buf_len", kernel boot argument "log_bug_len", **dynamically allocating** if larger than static buffer

2.6.11/2005-01: BKL changed to semaphore, **now call_console_drivers() must be called with interrupts disabled**

2.6.12/2005-03: **add timing information to messages**

2.6.25/2008-01: do not allow printk to recurse (unless oops_in_progress), stores last recursed message at the beginning of the buffer to console print

2.6.26/2008-05: **hide console printing latency from irq latency tracer**

2.6.27/2008-08: add printk tick to wake syslog

2.6.32/2009-10: **kmsg_dump interface**

2.6.35/2010-05: support for dmesg from kdb

2.6.36/2010-06: trigger console printing when a CPU comes online

2.6.39/2011-03: **add exclusive_console "feature" to avoid multiple messages**

3.3/2012-02: for scheduler context store in a per-cpu (single message) buffer, the next printk tick printk's it

A Brief History of printk

3.4/2012-05:	re-implement ringbuffer with **variable record structures** and **sequence numbers**, recursion messages stored in ringbuffer, **/dev/kmsg interface** (read and write)
3.6/2012-07:	change loglevel markers to SOH (start of header) character
3.7/2012-10:	remove printk tick, use irq_work to trigger syslog waking
3.15/2014-06:	for scheduler context store directly to ringbuffer, irq_work now only console prints, **report number of dropped messages**
3.18/2014-06:	add per-cpu printk function pointer for per-cpu diversion, use a per-cpu nmi_seq_buf to dump backtraces from NMI then call printk for all the nmi_seq_bufs
4.5/2016-01:	allow scheduler to run between lines when console printing (if console may schedule)
4.7/2016-05:	flush NMI buffers to ringbuffer on panic
4.10/2016-12:	**safe buffers**, per-cpu function replaced with per-cpu context variable, **used for NMI and recursion protection**, **flush safe buffers to ringbuffer on panic**, oops_in_progress no longer used to force logbuf_lock/console_sem re-init
4.12/2017-04:	store to ringbuffer from any context (if possible)
4.15/2018-01:	**add console owner/waiter logic to hand-off console printing**
5.0/2019-02:	finally clean **LOG_CONT ordering based on caller identifier**

Open Issues

ringbuffer protected by `raw_spinlock`

- requires use of safe buffers for some contexts

safe buffers

- bogus timestamps
- relies on irq-work mechanism
- cannot be sync'd on panic if CPUs do not go offline

console drivers

- possibly very slow
- all registered consoles called with interrupts disabled
- interrupt latencies ”ignored”
- unreliable in panic situation

Open Issues

`pr_info()` handled the same as `pr_emerg()`

- users are forced to restrict loglevels to avoid system latencies

last console-owner pays the highest price

- handing off console printing is great... unless you are the last one
- not unbounded but it is a big wildcard for callers of `printk()`

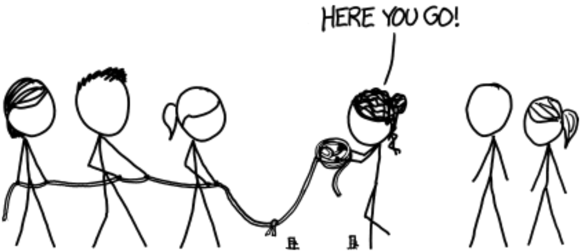
`oops_in_progress` and `bust_spinlocks()`

- we can do better than ignoring locking and hoping for the best

Main Issue: Tug-of-War

non-interference vs. reliability

"In the ultimate game of tug-of-war the only winning move is not to pull."



[2]

[2] CC BY-NC 2.5, Randall Munroe, https://what-if.xkcd.com/127

Main Issue: Tug-of-War

What does it mean ”not to pull”?

The two conflicting requirements can be differentiated:

- **what** is being printk’d
- **when** is it being printk’d

Split the problem into 2 problems (with 2 separate solutions):

- **non-interference**: make printk fully preemptible
  - all-context-safe ringbuffer
  - per-console kthreads
- **reliability**: provide an official synchronous channel for important messages
  - all-context-safe ringbuffer
  - atomic consoles
  - emergency messages

Ringbuffer (cpu-lock)

non-interference and reliability

Features

- concurrent multi-readers and single writing CPU
- supports all contexts
- all record data stored contiguously
- simple implementation[3]

Implementation

- uses a CPU-reentrant spinlock (cpu-lock) to serialize writers[4]
- uses logical positions to avoid ABA problem via tagged states

---

[3] see Appendix 1 for details
[4] see Appendix 2 for details

Ringbuffer (cpu-lock)

Concerns

- the cpu-lock has a "BKL feel" to it
  - there can only be 1 in the system
  - all NMI locking must be done under the cpu-lock

Ringbuffer (lockless)

non-interference and reliability

Features

- truly lockless
- concurrent multi-readers and multi-writers
- supports all contexts
- raw record data stored contiguously

Implementation

- uses descriptors to store meta-data of records
- uses a numbered list (numlist) to sequence the records
- uses a data ringbuffer (dataring) to manage raw record data
- high-level printk-ringbuffer to provide a coherent reader/writer interface to support the features of printk
- uses node IDs and logical positions to avoid ABA problem via tagged states

Ringbuffer (lockless)

Concerns

- complex
  - 3 different data structures[5]
  - multiple writer variables shared between CPUs
  - 9 memory barrier pairs
- difficult to document
  - lacking formalized memory barrier documentation guidelines[6]
- difficult to review
- Is multi-writer support really necessary?

---

[5] see Appendix 3 for details
[6] see Appendix 4 for details

Per-Console kthreads

non-interference

Features

- decouple `printk()` callers from the console
- individual iterators per console
  - let them go as fast as they can
  - allowing fast consoles to run fast can also help reliability (although any reliability would only be a bonus)
- individual loglevels per console
  - users could reduce printing for slow consoles and increase it for fast ones
- things like `exclusive_console` are solved automatically
  - each console has its own iterator

Per-Console kthreads

Concerns

- What about the console lock?
  Turn it into a rwlock where:
  - console-kthreads are readers (multiple allowed)
  - non-printk console lockers are writers (only 1 allowed)
    - Who exactly are these console lockers and what are they doing?
    - Could a per-console lock suffice?
  - `printk()` callers will not care about the console lock
- Can we rely on kthreads for console printing?
  - the emergency messages and atomic consoles
    solve the reliability problem
  - if we can learn to ”stop pulling” on each other, we can focus our energy on optimizing the two separate solutions

Atomic Consoles

reliability

Features

- new console callback `write_atomic()`
  - a safe early printk
- synchronous printing
  - "back to the roots" of printk
  - make the `printk()` caller do its own work
  - ignores console lock
    (synchronized at the console driver level using the cpu-lock)
- only prints "emergency messages"
- allows console drivers to be preemptible for PREEMPT_RT
  - no need to "ignore" console printing latencies anymore
- no need for special `oops_in_progress` handling or `bust_spinlocks()`

Atomic Consoles

Concerns

- How many console drivers could actually implement this?
  - RFCv1 implemented 8250 UART as an example
- the cpu-lock is needed for NMI safety
  - the console callbacks `write()` and `write_atomic()` must synchronize against each other
  - most console drivers need locks for printing (even UARTs)
- Can something be done for systems without atomic consoles?
  - provide a special console that writes to predictable memory addresses that may survive reboots
  - print synchronously in non-atomic context (if it can be detected)
  - on panic fallback to ”legacy” `oops_in_progress/bust_spinlocks()` behavior (not possible with PREEMPT_RT)

Emergency Messages

reliability

Features

- messages can be tagged for synchronous and immediate console printing

Concerns

- What is considered important?
- Should developers decide what an emergency message is?
  - `BUG()`, `WARN()`, oops, panic
  - console drivers decide based on some criteria/configuration
- Should users decide?
  - based on a loglevel threshold

Status (until now)

- work began 13 Feb 2018
  - with behind-the-scenes support from Peter Zijlstra, Thomas Gleixner, Steven Rostedt
- presented ”in progress” work at ELCE 2018 RT-Summit 25 Oct 2018
  - general feedback positive
- RFCv1 posted 12 Feb 2019
  - ringbuffer (cpu-lock), console printing thread, emergency messages, atomic consoles (with 8250 UART implementation)
  - feedback: poorly documented, too many changes at once, too many controversial changes
  - agreed upon a general roadmap to proceed
  - integrated into PREEMPT_RT since Linux 5.0.3-rt1
- RFCv2 posted 7 Jun 2019
  - only includes a ringbuffer (lockless) and test module
  - feedback: poorly documented, too complex, too monolithic
- RFCv3 posted 27 Jul 2019
  - ringbuffer from RFCv2 completely refactored (implementing numlist and dataring) and re-documented
- RFCv4 posted 8 Aug 2019
  - includes ringbuffer from RFCv3 and replaces ringbuffer in `printk.c`
  - feedback: possibly over-documented, still too complex

Status (looking forward)

1. replace the mainline ringbuffer
   - decide which ringbuffer to use!
   - keep `logbuf_lock` (for now)
   - (for lockless) refactor the code for simplification
   - (for lockless) formalize memory barrier comments
2. implement an NMI-safe `LOG_CONT`
3. remove `logbuf_lock`
4. remove safe buffers
   - allow 1 level of printk recursion?
5. implement per-console kthreads
   - possibly as optional mode of operation (like threadirqs now)
6. implement emergency messages
   - determine how emergency messages are classified
7. implement `write_atomic()` for 8250 UART driver
   - sort out cpu-lock concerns

NOTE: Code for nearly all of this was posted in RFCv1. Although it may not be suitable ”as is”, it functions quite well as a working prototype that already tackles the hard problems and can be tested in various scenarios.

Thank You

Many thanks to Petr Mladek, Peter Zijlstra, Sergey Senozhatsky, Andrea Parri, Thomas Gleixner, Linus Torvalds, and Greg Kroah-Hartman for positive and enlightening discussions!

Special thanks to the Linux Foundation for supporting our efforts to bring PREEMPT_RT mainline.

Questions / Comments

Thank you for your attention!

<john.ogness@linutronix.de>

A.1 Ringbuffer (cpu-lock)

Implementation

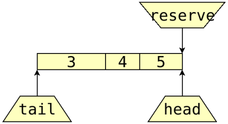
The cpu-lock ringbuffer works as follows:

- all operations (reserve/commit/free) are performed under the cpu-lock
- sequence numbers are assigned by the task/context that took ownership of the cpu-lock
- readers can only see records between the tail and head

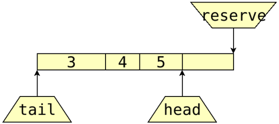
The following slides show the steps when adding and removing records for the cpu-lock ringbuffer.

A.1 Ringbuffer (cpu-lock)

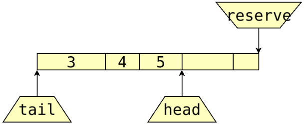
Steps of adding a record to the ringbuffer:



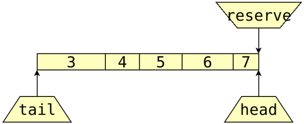
1. The ringbuffer initially contains 3 records.



2. Space for a new record is reserved by setting the reserve pointer using `cmpxchg()`.



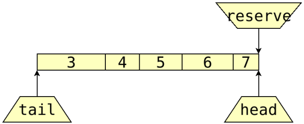
3. Assuming an NMI interrupt occurred, space for another record is reserved. This record is committed in NMI context, even though it does not yet have a sequence number.



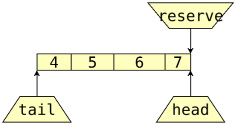
4. Upon commit of the first reserved space, the sequence numbers for all records up until the reserved pointer are set. The head pointer is updated using `cmpxchg()`.

A.1 Ringbuffer (cpu-lock)

Steps of removing a record from the ringbuffer:



1. The dataring initially contains 5 records.



2. The tail is set to record 4 using `cmpxchg()`.

A.2 CPU-Lock

Implementation

The cpu-lock is a CPU-reentrant spinlock. It works by:

- tracking which CPU currently owns the cpu-lock
- if a CPU already owns the lock, the lock function just returns
- each task/context tracks if it was the one to take ownership initially
- if a task/context did not take ownership, the unlock function just returns

The following slide shows the implementation of the cpu-lock.

A.2 CPU-Lock

```
struct prb_cpulock {
    atomic_t owner;
    unsigned long __percpu *irqflags;
};
```

```
void prb_lock(struct prb_cpulock *cpu_lock,
              unsigned int *cpu_store)
{
    unsigned long *flags;
    unsigned int cpu;

    for (;;) {
        cpu = get_cpu();

        *cpu_store = atomic_read(&cpu_lock->owner);
        if (*cpu_store == -1) {
            flags = per_cpu_ptr(cpu_lock->irqflags, cpu);
            local_irq_save(*flags);
            if (atomic_try_cmpxchg_acquire(&cpu_lock->owner,
                                           cpu_store, cpu)) {
                /* this CPU now owner */
                return;
            }
            local_irq_restore(*flags);
        } else if (*cpu_store == cpu) {
            /* this CPU is already owner */
            return;
        }

        put_cpu();
        cpu_relax();
    }
}
```

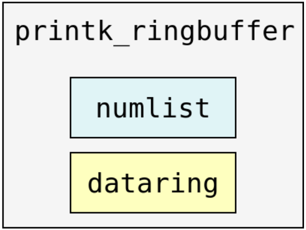
```
void prb_unlock(struct prb_cpulock *cpu_lock,
                unsigned int cpu_store)
{
    unsigned long *flags;
    unsigned int cpu;

    cpu = atomic_read(&cpu_lock->owner);
    atomic_set_release(&cpu_lock->owner, cpu_store);

    if (cpu_store == -1) {
        flags = per_cpu_ptr(cpu_lock->irqflags, cpu);
        local_irq_restore(*flags);
    }

    put_cpu();
}
```

A.3 Ringbuffer (lockless)

Internal Data Structures

The printk-ringbuffer is composed of:

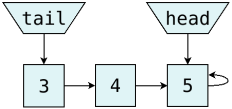
- numlist
  - manages the list of committed records
  - always sorted based on sequence numbers
- dataring
  - manages raw record data
  - sorted based on reserve order (not relevant)



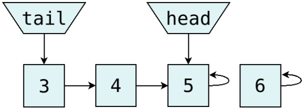
The following slides show the steps when adding and removing records for the numlist and dataring structures. After that, a slide shows some example relational scenarios between the numlist and dataring.

A.3 Ringbuffer (lockless)

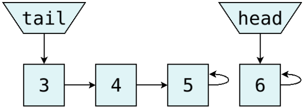
Steps of adding a node (record) to the numlist (committed list):



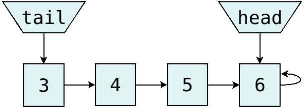
1. The committed list initially contains 3 records. Record 5 is the terminating record.



2. A new node (record 6) is setup as terminating.



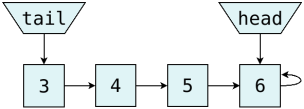
3. The head is set to record 6 using `cmpxchg()`.



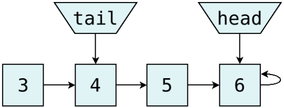
4. The next of record 5 is set to point to record 6.

A.3 Ringbuffer (lockless)

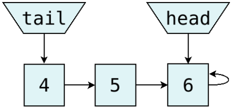
Steps of removing a node (record) from the numlist (committed list):



1. The committed list initially contains 4 records. Record 3 is the oldest record.



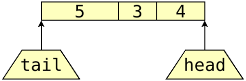
2. If record 3 is not pointing to valid data, the tail is set to record 4 using `cmpxchg()`.



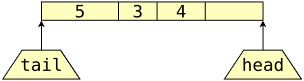
3. Record 3 was successfully removed and can be recycled.

A.3 Ringbuffer (lockless)

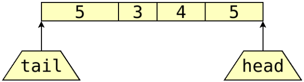
Steps of adding a datablock (record) to the dataring:



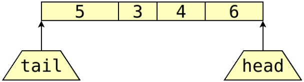
1. The dataring initially contains 3 records. Record 5 is the newest, record 3 the oldest.



2. A datablock (record 6) is reserved by setting the head using `cmpxchg()`.



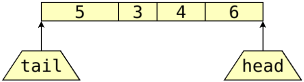
3. The newly reserved datablock is initially set with the wrong ID (5) until the writer commits the data.



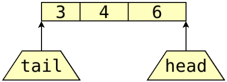
4. Upon committing, the datablock is set with the correct ID (6). Now it can be removed.

A.3 Ringbuffer (lockless)

Steps of removing a datablock (record) from the dataring:



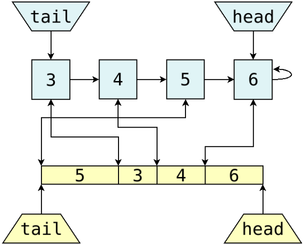
1. The dat1ring initially contains 4 records. Record 6 is the newest, record 3 the oldest. But record 5 will be removed.



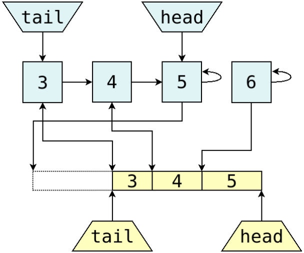
2. If datablock 5 points to a valid record, the tail is set to record 3 using `cmpxchg()`.

A.3 Ringbuffer (lockless)

Different examples of numlist/dataring relations. [7]



All 4 records (3, 4, 5, 6) belong to the committed list (i.e. are visible to readers) and point to valid data.



Records 3, 4, 5 are visible to readers, however the data for record 5 has been dropped. Record 6 has been reserved by a writer, but not yet committed. The ID for its data is purposely set to the wrong ID (5) until it has been committed by the writer.

[7] For simplicity, the ID and the sequence number are shown to be the same.

A.4 Memory Barriers

Usage in the Ringbuffer (lockless)

- splitting the ringbuffer into separate data structures helped to simplify the barriers, but they are still quite complex
- LKMM proofs performed using herd7 litmus tests
- attempted to formally document the memory barriers within the source code
  - no formal guidelines exist (yet?)

The following slide shows an example of using a litmus test to verify the memory barriers. The slide after that shows an example of trying to formalize the documentation of such memory barriers.

A.4 Memory Barriers

left column: pseudo code
right column: litmus test

```
void numlist_push(struct numlist *nl,
                  struct nl_node *n)
{
    unsigned long head_id;
    struct nl_node *head;
    u64 seq;

    head_id = READ_ONCE(nl->head_id);
    head = to_node(nl, head_id);

    seq = READ_ONCE(head->seq);

    n->seq = seq + 1;

    cmpxchg_release(&nl->head_id,
                    head_id, n->id);
}
```

The litmus test verifies that a pushed numlist node will always have a sequence number that is exactly +1 of the sequence number of the previous node.

```
{
    int node1 = 1;
    int *nl_head = &node1;
}
P0(int **nl_head, int *node1, int *node2)
{
    int r;

    *node2 = 2;
    r = cmpxchg_release(nl_head, node1, node2);
}
P1(int **nl_head)
{
    int *head;
    int seq;

    head = READ_ONCE(*nl_head);
    seq = READ_ONCE(*head);
}
exists (1:head=node2 /\ 1:seq!=2)
```

```
$ herd7 -conf linux-kernel.cfg seq_push.litmus
Test seq_push Allowed
States 2
1:head=node1; 1:seq=1;
1:head=node2; 1:seq=2;
No
Witnesses
Positive: 0 Negative: 2
Condition exists (1:head=node2 /\ not (1:seq=2))
Observation seq_push Never 0 2
Time seq_push 0.00
Hash=67e0375d84092a2f96833746fcff0500
```

A.4 Memory Barriers

```
void numlist_push(struct numlist *nl, struct nl_node *n, unsigned long id)
{
        unsigned long head_id;
        struct nl_node *head;
        u64 seq;

        /* LMM_TAG(A) */
        head_id = READ_ONCE(nl->head_id);
        head = to_node(nl, head_id);

        /* LMM_TAG(B) */
        seq = READ_ONCE(head->seq);

        /*
         * LMM_TAG(C)
         * Set @n->seq to +1 of @seq from the previous head.
         *
         * If LMM_REF(numlist_push:A) reads from LMM_REF(numlist_push:D),
         * then LMM_REF(numlist_push:B) reads from LMM_REF(numlist_push:C).
         *
         * Relies on:
         * RELEASE from LMM_REF(numlist_push:C) to LMM_REF(numlist_push:D)
         *     matching
         * ADDRESS DEP. from LMM_REF(numlist_push:A) to LMM_REF(numlist_push:B)
         */
        n->seq = seq + 1;

        /*
         * LMM_TAG(D)
         * Guarantee that @n->seq is stored before this node is visible
         * to any pushing writers. It pairs with the address dependency
         * between @head_id and @seq. See LMM_REF(numlist_push:C) for
         * details.
         */
        cmpxchg_release(&nl->head_id, head_id, id);
}
```