printk(). The Road Ahead.

OSS/KS
October 2017
1. What is this presentation

- This is “a slower” printk() talk
- This is a version 2 of “printk() considered harmful” talk (one year later)
  - Things we tried and lessons we learned
  - Things we still want to try
1. **Meanwhile in printk()...**

   - printk() is complicated
   - It’s not just a way to store your messages in a logbuf
   - Sometimes it is... sometimes it’s not
   - It also has a number of important locks
   - That number is… unknown. And depends on your .config
   - To make it even more fun, one can printk(), for example, from NMI
   - Which means that printk() can be interrupted and CPU can re-enter printk() while previous printk() is in unknown state
2. Deadlocks in printk()

- printk() from NMI is not an issue anymore
  - printk() from NMI is much better now, thanks to printk-nmi per-CPU buffers
- You still can deadlock your system in printk()
2. **Deadlocks in printk()**

- A simple case might look as follows:
  - Acquire run queue rq->lock
  - Invoke `printk()`
    - `printk()` uses `console_sem` semaphore
    - Which might invoke `wake_up_process()` from `up()`
    - Which acquires `rq->lock`
2. Deadlocks in printk()

- A harder case:
  - lockdep warning which recurses back into printk()
- That’s why printk() used to disable lockdep via lockdep_off()
- The problem with lockdep_off() is that... it disables lockdep validator
- And it also disables RCU validator
  - Which is bad enough
2. **Deadlocks in printk()**

- An even harder case:
  - WARN_ON()/BUG_ON()/dump_stack() which recurses back into printk()
- We don’t have any solution here yet
- The simplest thing is to switch WARN_ON/etc to printk-safe
- But printk-safe has that “will print soon” thing, which might look scary (not soon enough)
2. Deadlocks in printk()

- What we did to improve printk() was something that we called safe printk (printk_safe)
  - Basically it’s the same idea as NMI printk()
  - Additional per-CPU buffers to redirect unsafe printk() output to
  - We switch between printk_safe and normal printk() modes via printk_safe_enter()/printk_safe_exit() calls
  - We flush printk_safe buffers from IRQ work
2. Deadlocks in printk()

- The GOOD:
  - We made printk() reentrant
  - We keep lockdep enabled in printk() code
    - This has already revealed a number of bugs that were previously hidden
  - We did some panic printk() cleanups (killed zap_locks())
  - We made printk() less deadlock prone
2. Deadlocks in printk()

- The BAD:
  - printk(), frankly, is not quite reentrant
  - We use 2 buffers for NMI printk() and printk_safe on each CPU
    - printk_safe does not share buffers with NMI printk()
    - Because NMI can interrupt printk_safe, which will result in lost NMI messages
  - We need to manually switch between printk_safe, NMI printk(), normal printk() modes
    - Not an issue with NMI printk, but printk_safe is different
  - We can’t use printk_safe from a sleeping context
    - IOW it’s OK to switch to printk_safe for up() or down_trylock()
    - But it’s a NO-NO for down()
2. Deadlocks in printk()

- The UGLY:
  - printk(), in fact, is not reentrant
  - printk() can still deadlock
  - We don’t have a solution that would make everyone happy
2. Deadlocks in printk()

SyS_ioctl
tty_ioctl
tty_mode_ioctl
uart_set_termios
uart_change_speed
FOO_serial_set_termios
    spin_lock_irqsave(&port->lock) // lock the output port
/* WARN_ON() / BUG_ON() / printk() */
vprintk_emit()
console_unlock()
call_console_drivers()
FOO_write()
    spin_lock_irqsave(&port->lock) // deadlock
2. **Deadlocks in printk()**

- The fundamental issue is that `printk()` depends on two different lock types - internal and external locks.
- We can handle internal locks (`console_sem`, `logbuf spin_lock`) with `printk_safe`.
- External locks are out of control: console locks, scheduler locks, timekeeping locks, etc.
2. **Deadlocks in printk()**

- A possible solution might sound like - `printk_deferred()` everywhere
  - Deferred `printk()` just appends message to the logbuf and then queues IRQ work to print pending messages sometime later
  - IOW, we effectively remove all external locks and deal with internal `printk()` locks only… just one lock, in fact - `logbuf` lock
  - But this solution has its drawbacks and limitations, tho
  - Namely, we need to guarantee that IRQ work will emit messages eventually
    - So really bad-bad hard lockup cases won’t work
    - On UP system any hard lockup is a “bad-bad thing”
  - (There is a “one more thing” here)
2. **Deadlocks in printk()**

- Another proposal (PeterZ) is to have `early_printk` fallback
- Which avoids all the locking mess, but breaks dmesg and friends
- Surely, this solution is not for everyone
- But, at the same time, it definitely does what we want
2. **Deadlocks in printk()**

- We need to remove all (or at least to minimize the number of) external locks to make panic print out more likely
  - Especially NMI panic
- One more idea is to extend `struct console` and to introduce a new callback which we will call from panic handler
  - `con->write()` is for normal write, `con->write_on_panic()` is for very special cases
- That new callback ideally should be lockless (!)
  - We will call it from `panic()` handler
  - “Barely legal lockless”
2. **Deadlocks in printk()**

- This will basically combine two things in console drivers
- A “normal” write
- And some sort of an “early” write mode (when in panic)
- It’s totally OK to have a small number of console drivers supporting `write_on_panic()`
2. Deadlocks in printk()

- We used to have a `zap_locks()` function in `printk()`:
  - Which would simply re-init `printk()` internal locks when we saw a recursive `printk()` during panic
  - But `printk()` locks are, in fact, very small part of the problem
  - When `zap_locks()` was introduced we didn’t have a better idea on how to handle `printk()` recursion. We do have now, tho
  - So we removed `zap_locks()` from `printk()`
2. Deadlocks in printk()

- Console drivers are not re-entrant
- Alternatively, maybe, we can extend struct console and add per-console ->zap_locks()
  - Which we will call from panic()
  - ->zap_locks() would re-init console driver locks and make it possible to re-enter the console driver ->write() function
2. Deadlocks in printk()

- Console drivers are black boxes
- Maybe we can factor out console driver locking (e.g. port locking for UART drivers)
- Let’s say ->lock() and ->unlock() (we need to extend struct console, once again)
- Call con->lock()/con->unlock() in the normal printing loop
- Don’t call con->lock()/con->unlock() when the system is in panic
## 2. Deadlocks in printk()

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
</table>
| avoid call_console_drivers(....)  
   {  
     for_each_console(con) {  
       if (!(con->flags & CON_ENABLED))  
         continue;  
       con->write(con, text, len);  
     }  
   }  
| void call_console_drivers(....)  
   {  
     for_each_console(con) {  
       if (!(con->flags & CON_ENABLED))  
         continue;  
       con->lock(con);  
       con->write(con, text, len);  
       con->unlock(con);  
     }  
   } |
3. **Console semaphore**

- Remember when I said that `console_sem` was an internal `printk()` lock?
  - It’s … actually not
  - `printk()` is just one of its users
- `console_sem`, on the `printk()` side:
  - Ensures that there is only one printing CPU
    - We do printing from `console_unlock()`
  - Protects console drivers list
  - Protects console drivers
3. Console semaphore

- On the console drivers’ side:
- `console_sem` is used to synchronize different types of events
  - `printk()` vs TTY
    - Because writing to console is quite difficult: you need to handle scrolling, wrap the lines, UTF8/ASCII chars, control characters like \r or \n, etc.
    - You don’t want `printk()` and TTY to mix
  - Timers (e.g. cursor blinking)
  - IRQs
    - You don’t want `printk()` to race with `printk()` from IRQ
  - I/OCTLS
    - You don’t want to resize console while `printk()` or TTY are actively printing to it
  - Notifiers (including PM)
  - etc.
3. Console semaphore

- DRM/KMS/FBCON/etc. need to acquire `console_sem` for things not directly related to printing
- These things are called from different contexts
  - Some of `console_sem` owners can schedule
    - `console_lock(); mutex_lock(&foo); console_unlock();`
    - `console_lock(); kmalloc(GFP_KERNEL); console_unlock();`
- Even more
  - `printk()` can schedule from `console_unlock()`
3. Console semaphore

- console_sem can be part of livelock scenarios, which will prevent any kernel logs from appearing on the consoles.
- A small example (CPU1 couldn’t make any OOM progress):

<table>
<thead>
<tr>
<th>CPU0</th>
<th>CPU1</th>
<th>CPU2</th>
</tr>
</thead>
<tbody>
<tr>
<td>printk()</td>
<td>mutex_lock(&amp;par-&gt;bo_mutex)</td>
<td>console_callback()</td>
</tr>
<tr>
<td>vprintk.emit()</td>
<td>kzalloc(GFP_KERNEL)</td>
<td>console_lock()</td>
</tr>
<tr>
<td>if (console_trylock())</td>
<td>kmem_cache_alloc()</td>
<td>mutex_lock(&amp;par-&gt;bo_mutex)</td>
</tr>
<tr>
<td>console_unlock();</td>
<td>io_schedule_timeout()</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
3. Console semaphore

- IOW, if you have something going on in one of the console drivers (something that requires \texttt{console\_lock()})
- Or something is going on in the current \texttt{console\_sem} owner context (preemption)
- Then there will be no \texttt{printk()} output to any of the console drivers at all
  - As long as \texttt{console\_sem} is locked
  - Or until the system panics and invokes \texttt{flush\_on\_panic()}
    - Which ignores \texttt{console\_sem} state
- And this is why \texttt{printk\_deferred()} offloading is not reliable
  - We still do \texttt{console\_trylock()} there
3. Console semaphore

- We sort of do and don’t have a plan at the same time
- It’s a long-long way to go in any case
- What we call `printk()` is in fact a huge mix of different subsystems: framebuffers, serial consoles, TTY, sched, timekeeping, networking, etc.
3. Console semaphore

- **WARNING**: this might be as “good” as immediate `printk()` offloading
- Maybe it’s time to introduce new `printk()` API
- Switch `printk()` to a polling mode
  - Each registered console driver would poll `logbuf` and print only unseen messages
    - Per-console `console_seq` flag
    - So console will do something like
      - `con->console_seq = logbuf_get_message_at(con->console_seq, buffer)`
  - We, in fact, sort of already *(not for the kernel logs!)* do this in UART
    - Serial drivers print up to N (or all) pending xmit chars from IRQ handlers
  - This way messages would appear on a particular console driver only when that driver is ready to print the messages
3. **Console semaphore**

- The interesting thing is that *(some of)* polling consoles ideally won’t depend on `console_sem`
  - `console_sem` may be locked forever by a misbehaving console, or suffer from a livelock scenario, etc.
- Polling consoles should not care
  - Some of them can’t, in fact, acquire `console_sem` because polling is done from the IRQ handler (UART)
3. Console semaphore

- Weak points:
  - No direct `printk()` flush
  - Hard Lockups are still a massive pain
  - NMI `printk()` is a massive pain
4. **Yet another way to kill your system**

- The way `printk()` designed is that there is always one CPU doing the printing job.
- Other CPUs simply append messages to the logbuf.
- Printing CPU does not stop until there are no pending logbuf messages left.
- This is known to cause all sorts of problems: lockups, OOMs, stalls, etc.
- To mitigate some of those problems rescheduling points have been added to the `console_unlock()` function.
- At some point we extended the number of rescheduling cases even further.
4. Yet another way to kill your system

- This, however, didn’t fix everything - printing from atomic context, etc.
- At the same time, while rescheduling makes watchdog happy it slows down `printk()` and this can be problematic
  - For instance, OOM print out can take many seconds in some cases, pushing the kernel further towards the OOM wall
4. **Yet another way to kill your system**

- We now have both
  - Requests from people to backport those changes to stable kernels
  - And at the same time complaints and bug-reports from other people
- Even more worse, rescheduling with locked `console_sem` was not a good decision on our side
  (my personal opinion)
4. Yet another way to kill your system

- So one year ago we had a solution which looked simple and reasonable: avoid direct printk() flush as much as possible and do printk() offloading. Immediately.
- printk() would simply log_store() and then wake up a dedicated printk_kthread to do the actual printing (unless in panic)
- We tried it out, we saw a number of problems, we received a bunch of complaints from other people
  - Long story short, I didn’t like it
4. Yet another way to kill your system

- First, people do want to have direct `printk()`
  - Especially when the systems locks up in a peculiar way shortly after `printk()`
    - Things like CPU stop IPIs need to be done in direct flush `printk()` mode
  - Relying on the scheduler and `printk_kthread` is not an option
  - Thus offloading straight ahead does not look attractive
4. Yet another way to kill your system

- Our new approach is a “postponed offloading” - give the task a chance to perform direct printk(), but prevent it from locking up the system
- We used to have a user defined timeout value (sysfs knob)
- But it’s really a watchdog lockup threshold value that matters
  - Or RCU stall detection threshold
  - (we don’t have spinlock lockup timeout loop anymore)
Yet another way to kill your system

- The weak point is that we leave `console_unlock()` with pending logbuf messages
- We `wake_up()` printk_kthread, but it’s unclear when it will take over, if at all
- After a number of unsuccessful offloading attempts `printk()` switches to emergency mode and stops offloading attempts
4. **Yet another way to kill your system**

- Of course, there are cases when we can’t offload
  - Suspend, kexec, panic, etc.
- For such cases we provide a new API that temporarily disables printk() offloading
- There is “one more thing” - sysrq print out
  - It’s lengthy, time consuming…. and important
  - Thus sysrq print out usually forcibly suppresses watchdog warnings (touch watchdog functions)
  - We can’t offload in those cases as well
4. Yet another way to kill your system

- Offloading helps us to land other improvements
- If we know that offloading is enabled we can avoid rescheduling from `console_unlock()`
  - `console_sem` owner is now busy doing what it has to do - print out pending messages
- We can do so because all processes, including `printk_kthread`, unlock `console_sem` periodically
- `printk_kthread` attempts to re-acquire the lock again (if there are pending logbuf messages)
  - So it's either `printk_kthread` or one of `console_sem` waiters that will continue printing
4. Yet another way to kill your system

- Why do we do this?
- Because processes that are blocked on console_sem sleep in TASK_UNINTERRUPTIBLE
  - Including user-space processes (systemd, mount, etc.)
- Unlocking console_sem will basically let those processes to `wake_up()` and do some useful work
4. Yet another way to kill your system

- An alternative solution (proposed two days ago by Steven Rostedt)
- A sort of round-robin print out
- If there is a CPU doing printing (looping in `console_unlock()`)
- CPU that calls `printk()` will mark itself as a waiter for the `console_sem`
- The printing CPU will detect that there is a new task willing to take over
- It will then `up()` the `console_sem` so the `printk()` waiter, probably, will lock the `console_sem` and continue printing
  - There are scenarios when this probably won’t do the trick
  - Need to do more review/testing/etc.
Thank you.