LinuxInternals.org

Tinkering with Embedded Linux

- RSS

Search

Navigate… ▼

- Blog
- Archives
- Resources
- About me

Linux Spinlock Internals

May 7th, 2014 | Comments

This article tries to clarify how spinlocks are implemented in the Linux kernel and how they should be used correctly in the face of preemption and interrupts. The focus of this article will be more on basic concepts than details, as details tend to be forgotten more easily and shouldn't be too hard to look up although attention is paid to it to the extent that it helps understanding.

Fundamentally somewhere in `include/linux/spinlock.h`, a decision is made on which spinlock header to pull based on whether SMP is enabled or not:

```
#if defined(CONFIG_SMP) || defined(CONFIG_DEBUG_SPINLOCK)
# include <linux/spinlock_api_smp.h>
#else
# include <linux/spinlock_api_up.h>
#endif
```

We'll go over how things are implemented in both the SMP (Symmetric Multi-Processor) and UP (Uni-Processor) cases.

For the SMP case, `__raw_spin_lock*` functions in `kernel/locking/spinlock.c` are called when one calls some version of a `spin_lock`.

Following is the definition of the most basic version defined with a macro:

```
#define BUILD_LOCK_OPS(op, locktype)                                    \
void __lockfunc __raw_##op##_lock(locktype##_t *lock)                   \
{                                                                       \
        for (;;) {                                                      \
 5                 preempt_disable();                                  \
 6                 if (likely(do_raw_##op##_trylock(lock)))            \
 7                         break;                                      \
 8                 preempt_enable();                                   \
 9                                                                     \
10                 if (!(lock)->break_lock)                            \
11                         (lock)->break_lock = 1;                     \
12                 while (!raw_##op##_can_lock(lock) && (lock)->break_lock)\
13                         arch_##op##_relax(&lock->raw_lock);         \
14         }                                                           \
15         (lock)->break_lock = 0;                                     \
16 }                                                                   \
```

The function has several imporant bits. First it disables preemption on line 5, then tries to *atomically* acquire the spinlock on line 6. If it succeeds it breaks from the `for` loop on line 7, leaving preemption disabled for the duration of crticial section being protected by the lock. If it didn't succeed in acquiring the lock (maybe some other CPU grabbed the lock already), it enables preemption back and spins till it can acquire the lock keeping *preemption enabled during this period*. Each time it detects that the lock can't be acquired in the `while` loop, it calls an architecture specific relax function which has the effect executing some variant of a `no-operation` instruction that causes the CPU to execute such an instruction efficiently in a lower power state. We'll talk about the `break_lock` usage in a bit. Soon as it knows the lock is free, say the `raw_spin_can_lock(lock)` function returned 1, it goes back to the beginning of the `for` loop and tries to acquire the lock again.

What's important to note here is the reason for keeping preemption enabled (we'll see in a bit that for UP configurations, this is not done). While the kernel is spinning on a lock, other processes shouldn't be kept from preempting the spinning thread. The lock in these cases have been acquired on a *different CPU* because (assuming bug free code) it's impossible the current CPU which is trying to grab the lock has already acquired it, because preemption is disabled on acquiral. So it makes sense for the spinning kernel thread to be preempted giving others CPU time. It is also possible that more than one process on the current CPU is trying to acquire the same lock and spinning on it, in this case the kernel gets continuously preempted between the 2 threads fighting for the lock, while some other CPU in the cluster happily holds the lock, hopefully for not too long.

That's where the `break_lock` element in the lock structure comes in. Its used to signal to the lock-holding processor in the cluster that there is someone else trying to acquire the lock. This can cause the lock to released early by the holder if required.

Now lets see what happens in the UP (Uni-Processor) case.

Believe it or not, it's really this simple:

```
#define ___LOCK(lock) \
  do { __acquire(lock); (void)(lock); } while (0)

#define __LOCK(lock) \
  do { preempt_disable(); ___LOCK(lock); } while (0)

// ....skipped some lines.....
#define _raw_spin_lock(lock)                    __LOCK(lock)
```

All that needs to be done is to disable preemption and acquire the lock. The code really doesn't do anything other than disable preemption. The references to the `lock` variable are just to suppress compiler warnings as mentioned in comments in the source file.

There's no spinning at all here like the UP case and the reason is simple: in the SMP case, remember we had agreed that while a lock is *acquired* by a particular CPU (in this case just the 1 CPU), no other process on that CPU should have acquired the lock. How could it have gotten a chance to do so with preemption disabled on that CPU to begin with?

Even if the code is buggy, (say the same process tries to acquires the lock twice), it's still impossible that 2 *different processes* try to acquire the same lock on a Uni-Processor system considering preemption is disabled on lock acquiral. Following that idea, in the Uni-processor case, since we are running on only 1 CPU, all that needs to be done is to disable preemption, since the fact that we are being allowed to disable preemption to begin with, means that no one else has acquired the lock. Works really well!

Sharing spinlocks between interrupt and process-context

It is possible that a critical section needs to be protected by the same lock in both an interrupt and in non-interrupt (process) execution context in the kernel. In this case `spin_lock_irqsave` and the `spin_unlock_irqrestore` variants have to be used to protect the critical section. This has the effect of disabling interrupts on the executing CPU. Imagine what would happen if you just used `spin_lock` in the process context?

Picture the following:

1. Process context kernel code acquires *lock A* using `spin_lock`.
2. While the lock is held, an interrupt comes in on the same CPU and executes.
3. Interrupt Service Routing (ISR) tries to acquire *lock A*, and spins continuously waiting for it.
4. For the duration of the ISR, the Process context is blocked and never gets a chance to run and free the lock.
5. Hard lock up condition on the CPU!

To prevent this, the process context code needs call `spin_lock_irqsave` which has the effect of disabling interrupts on that particular CPU along with the regular disabling of preemption we saw earlier *before* trying to grab the lock.

Note that the ISR can still just call `spin_lock` instead of `spin_lock_irqsave` because interrupts are disabled anyway during ISR execution. Often times people use `spin_lock_irqsave` in an ISR, that's not necessary.

Also note that during the executing of the critical section protected by `spin_lock_irqsave`, the interrupts are only disabled on the executing CPU. The same interrupt can come in on a different CPU and the ISR will be executed there, but that will not trigger the hard lock condition I talked about, because the process-context code is not blocked and can finish executing the locked critical section and release the lock while the ISR spins on the lock on a different CPU waiting for it. The process context does get a chance to finish and free the lock causing no hard lock up.

Following is what the `spin_lock_irqsave` code looks like for the SMP case, UP case is similar, look it up. BTW, the only difference here compared to the regular `spin_lock` I described in the beginning are the `local_irq_save` and `local_irq_restore` that accompany the `preempt_disable` and `preempt_enable` in the lock code:

```
#define BUILD_LOCK_OPS(op, locktype)                                    \
unsigned long __lockfunc __raw_##op##_lock_irqsave(locktype##_t *lock)  \
{                                                                       \
        unsigned long flags;                                            \
                                                                        \
        for (;;) {                                                      \
                preempt_disable();                                      \
                local_irq_save(flags);                                  \
                if (likely(do_raw_##op##_trylock(lock)))                \
                        break;                                          \
                local_irq_restore(flags);                               \
                preempt_enable();                                       \
                                                                        \
                if (!(lock)->break_lock)                                \
                        (lock)->break_lock = 1;                         \
                while (!raw_##op##_can_lock(lock) && (lock)->break_lock)\
                        arch_##op##_relax(&lock->raw_lock);             \
        }                                                               \
        (lock)->break_lock = 0;                                         \
        return flags;                                                   \
}                                                                       \
```

Hope this post made a few things more clear, there’s a lot more to spinlocking. A good reference is Rusty’s Unreliable Guide To Locking.

Posted by Joel Fernandes May 7th, 2014

Tweet

« Studying cache-line sharing effects on SMP systems MicroSD card remote switch »

Comments

Featured Comment

raghu • 6 months ago

Hey Fantastic blog.
I have couple of questions.
1) In case of UP how is jiffies(timer) get updated while holding spin lock using spinlock_irq_save?
2)My understanding is in smp environment jiffies updation cannot be done on processor holding spinlock irrespective of spinlock API.So timer interrupt should be enabled across all cores in SoC. Imagine a scenario in quad core processor where 4 cores are holding 4 different spinlocks & how is the timer interrupt for jiffies handled?

1 ^ | ⌄ • Share ›

8 Comments Linux Internals 1 Login

Recommend 2 Share Sort by Best

Join the discussion…

raghu • 6 months ago

Featured by Linux Internals

Hey Fantastic blog.
I have couple of questions.
1) In case of UP how is jiffies(timer) get updated while holding spin lock using spinlock_irq_save?
2)My understanding is in smp environment jiffies updation cannot be done on processor holding spinlock irrespective of spinlock API.So timer interrupt should be enabled across all cores in SoC. Imagine a scenario in quad core processor where 4 cores are holding 4 different spinlocks & how is the timer interrupt for jiffies handled?

1 • Reply • Share ›

joelagnel Mod → raghu • 6 months ago

Glad you liked the article!
A timer interrupt before the next successive jiffie is not necessarily required. Whenever the CPU doing the update has its timer interrupt delivered, it uses the current time ('now' variable) to calculate how many ticks elapsed since the last jiffie update and does the new update.
See:
http://lxr.free-electrons.com/...

1 • Reply • Share ›

raghu → joelagnel • 6 months ago

Hi Joel
Thanks for responding.
But how is 'now' variable updated?

Scenario in UP:
spinlock_Irq_save gets lock just before PIT gives interrupt, since preemption & interrupt both disabled, the interrupt is latched & when interrupt enabled the jiffies get updated & PIT is reprogrammed by its ISR but with "delta delay" because ISR is called once spinlock_irq_restore.
my confusion is if interrupt gets latched & serviced bit later, is 'delta delay' accounted or some heuristic takes of this?

Hope I am not naive & asking very basic question.
Thanks & Regards
Raghu

1 ^ | ⌄ • Reply • Share ›

joelagnel Mod ↪ raghu • 6 months ago

Hey Raghu, The kernel has a clocksource that is continously getting updated in the hardware even when interrupts are disabled, this gives the kernel a sense of what is the current time since this clocksource was started, and from that it can calculate how much time has elapsed since the latest timer interrupt was serviced. When it gets a chance to update jiffies, "now" is calculated by reading this clocksource. From this, the delta delay can be easily calculated.
See:
http://lxr.free-electrons.com/...
where it calls ktime_get()

1 ^ | ⌄ • Reply • Share ›

raghu ↪ joelagnel • 6 months ago

Thanks Joel for explaining ....

^ | ⌄ • Reply • Share ›

pkbansal • a year ago

Amazing! It cant be explained better. Please keep up the good work.

1 ^ | ⌄ • Reply • Share ›

prasad m • 24 days ago

Hi ,
its awesome post its picture clear Thanks a lot
Thanks & Regards
Prasad.m

^ | ⌄ • Reply • Share ›

Kundan Kumar • 10 months ago

One more explaination :
http://sklinuxblog.blogspot.in...

^ | ⌄ • Reply • Share ›

ALSO ON **LINUX INTERNALS**

A MicroSD Card Remote Switcher

7 comments • 2 years ago

joelagnel — Glad you liked the header design. You can also find an article on hack-a-day on

Studying Cache-line Sharing Effects on SMP Systems

1 comment • 2 years ago

MRR — Cool shit! Keep writing

Recent Posts

- TIF_NEED_RESCHED: Why Is It Needed
- Tying 2 Voltage Sources/signals Together
- MicroSD Card Remote Switch
- Linux Spinlock Internals
- Studying Cache-line Sharing Effects on SMP Systems