

Re: [patch 2/5] [PREEMPT_RT] Changing interrupt handlers from running in thread to hardirq and back runtime.

Re: [patch 2/5] [PREEMPT_RT] Changing interrupt handlers from running in thread to hardirq and back runtime.

Source: <http://linux.derkeiler.com/Mailing-Lists/Kernel/2006-06/msg01029.html>

- *From:* Esben Nielsen <nielsen.esben@xxxxxxxxxxxxxxxx>
 - *Date:* Sun, 4 Jun 2006 18:33:35 +0100 (BST)
-

On Sat, 3 Jun 2006, Steven Rostedt wrote:

Disclaimer: I haven't read all your patches yet. I'm going one at a time to comment, and then I will probably send more emails about the overall response. So now, my comments are not on the big picture, but simple atomic views.

On Fri, 2006-06-02 at 23:23 +0100, Esben Nielsen wrote:

This patch makes it possible to change which context the interrupt handlers for each interrupt run in, hard-irq or threaded if CONFIG_PREEMPT_HARDIRQS is set.

The interface is the file
/proc/irq/<irq number>/threaded
You can read it to see what the context is now or write one of

```
irq
fifo <rt priority>
rr <rt priority>
normal <nice value>
batch <nice value>
```

where one of the latter makes the interrupt handler threaded.

A replacement for request_irq(), called request_irq2(), is added. When a driver

request_irq2 ... yuck! Perhaps request_irq_convertible() or something similar? But irq2, no way!

I know...

Re: [patch 2/5] [PREEMPT_RT] Changing interrupt handlers from running in thread to hardirq and back runtime

Re: [patch 2/5] [PREEMPT_RT] Changing interrupt handlers from running in thread to hardirq and back runtime.

The problem is:

- 1) Don't change existing drivers.
- 2) The `change_irq_context()` call-back must be set at `request_irq()`, so it is not enough to just make a new function where the driver can set it's the callback after the `request_irq()`.

`request_irq_convertible()` might be an ok name.

uses this to install it's irq-handler it can also give a `change_irq_context` call-back. This call-back is called whenever the irq-context is changed or going to be changed. The call-back must be of the form

```
int driver_change_context(int irq, void *dev_id, enum change_context_cmd cmd)
```

Eeee, looks like a big change to make on drivers, and something that can keep -rt from mainline forever. But I'll see more as I read. This looks optional but still it will make things more complex.

It is only optional, but the function is very easy to make.

where

```
enum change_context_cmd {  
    IRQ_TO_HARDIRQ,  
    IRQ_CAN_THREAD,  
    IRQ_TO_THREADED  
};
```

The call-back is supposed to do the following on

`IRQ_TO_HARDIRQ`: make sure everything in the interrupt handler is non-blocking

or return a non-zero error code.

`IRQ_CAN_THREAD`: Return 0 if it is ok for the interrupt handler to be run in

thread context. Return a non-zero error code otherwise.

`IRQ_TO_THREAD`: Now the interrupt handler does run in thread context.

The

driver can now change it's locks to being mutexes. The return

value is ignored as the driver already got a chance to protest above.

Re: [patch 2/5] [PREEMPT_RT] Changing interrupt handlers from running in thread to hardirq and back runtime

Re: [patch 2/5] [PREEMPT_RT] Changing interrupt handlers from running in thread to hardirq and back runtime.

Index: linux-2.6.16-rt23.spin_mutex/include/linux/interrupt.h

```
=====
--- linux-2.6.16-rt23.spin_mutex.orig/include/linux/interrupt.h
+++ linux-2.6.16-rt23.spin_mutex/include/linux/interrupt.h
@@ -34,21 +34,38 @@ typedef int irqreturn_t;
#define IRQ_HANDLED (1)
#define IRQ_RETVAL(x) ((x) != 0)

+enum change_context_cmd {
+ IRQ_TO_HARDIRQ,
+ IRQ_CAN_THREAD,
+ IRQ_TO_THREADED
+};
+
+struct irqaction {
+irqreturn_t (*handler)(int, void *, struct pt_regs *);
+unsigned long flags;
+cpumask_t mask;
+const char *name;
+void *dev_id;
+#ifdef CONFIG_CHANGE_IRQ_CONTEXT
+ int (*change_context)(int, void *,
+ enum change_context_cmd);
+#endif
+struct irqaction *next;
+int irq;
- struct proc_dir_entry *dir, *threaded;
+ struct proc_dir_entry *dir;
+ struct rcu_head rcu;
+};

extern irqreturn_t no_action(int cpl, void *dev_id, struct pt_regs *regs);
extern int request_irq(unsigned int,
irqreturn_t (*handler)(int, void *, struct pt_regs *),
unsigned long, const char *, void *);
+extern int request_irq2(unsigned int irq,
+ irqreturn_t (*handler)(int, void *, struct pt_regs *),
+ unsigned long irqflags, const char * devname,
+ void *dev_id,
+ int (*change_context)(int, void *,
+ enum change_context_cmd));
extern void free_irq(unsigned int, void *);
```

Index: linux-2.6.16-rt23.spin_mutex/include/linux/irq.h

```
=====
--- linux-2.6.16-rt23.spin_mutex.orig/include/linux/irq.h
+++ linux-2.6.16-rt23.spin_mutex/include/linux/irq.h
@@ -47,6 +47,18 @@
# define SA_NODELAY 0x01000000
#endif
```

Re: [patch 2/5] [PREEMPT_RT] Changing interrupt handlers from running in thread to hardirq and back runtime

Re: [patch 2/5] [PREEMPT_RT] Changing interrupt handlers from running in thread to hardirq and back runtime.

```
+#ifndef SA_MUST_THREAD
+# define SA_MUST_THREAD 0x02000000
+#endif
+
+/* Set this flag if the irq handler must thread under preempt-rt otherwise not
+ */
+#ifdef CONFIG_PREEMPT_RT
+# define SA_MUST_THREAD_RT SA_MUST_THREAD
+#else
+# define SA_MUST_THREAD_RT 0
+#endif
+
+/*
+ * IRQ types
+ */
+@@ -147,12 +159,13 @@ struct irq_type {
+ * @chip: low level hardware access functions – comes from type
+ * @action: the irq action chain
+ * @status: status information
+ * (protected by the spinlock )
+ * @depth: disable-depth, for nested irq_disable() calls
+ * @irq_count: stats field to detect stalled irqs
+ * @irqs_unhandled: stats field for spurious unhandled interrupts
+ * @thread: Thread pointer for threaded preemptible irq handling
+ * @wait_for_handler: Waitqueue to wait for a running preemptible handler
+ * @lock: locking for SMP
+ * @lock: lock around the action list
+ * @move_irq: Flag need to re-target interrupt destination
+ *
+ * Pad this out to 32 bytes for cache and indexing reasons.
+Index: linux-2.6.16-rt23.spin_mutex/kernel/Kconfig.preempt
+=====
+--- linux-2.6.16-rt23.spin_mutex.orig/kernel/Kconfig.preempt
++++ linux-2.6.16-rt23.spin_mutex/kernel/Kconfig.preempt
+@@ -119,6 +119,17 @@ config PREEMPT_HARDIRQS
```

Say N if you are unsure.

```
+config CHANGE_IRQ_CONTEXT
+ bool "Change the irq context runtime"
+ depends on PREEMPT_HARDIRQS && (PREEMPT_RCU ||
+ !SPIN_MUTEXES)
+ help
+ You can change whether the IRQ handler(s) for each IRQ number is
+ running in hardirq context or as threaded by writing to
+ /proc/irq/<number>/threaded
+ If PREEMPT_RT is selected the drivers involved must be ready for it,
+ though, or the write will fail. Remember to switch on SPIN_MUTEXES as
+ well in that case as the drivers which are ready uses spin-mutexes.
+
```

Re: [patch 2/5] [PREEMPT_RT] Changing interrupt handlers from running in thread to hardirq and back runtime.

Re: [patch 2/5] [PREEMPT_RT] Changing interrupt handlers from running in thread to hardirq and back runtime.

```
config SPINLOCK_BKL
bool "Old-Style Big Kernel Lock"
depends on (PREEMPT || SMP) && !PREEMPT_RT
Index: linux-2.6.16-rt23.spin_mutex/kernel/irq/internals.h
=====
--- linux-2.6.16-rt23.spin_mutex.orig/kernel/irq/internals.h
+++ linux-2.6.16-rt23.spin_mutex/kernel/irq/internals.h
@@ -22,6 +22,10 @@ static inline void end_irq(irq_desc_t *d
}

#ifdef CONFIG_PROC_FS
#ifdef CONFIG_CHANGE_IRQ_CONTEXT
+extern int make_irq_nodelay(int irq, struct irq_desc *desc);
+extern int make_irq_threaded(int irq, struct irq_desc *desc);
#endif
extern void register_irq_proc(unsigned int irq);
extern void register_handler_proc(unsigned int irq, struct irqaction *action);
extern void unregister_handler_proc(unsigned int irq, struct irqaction
*action);
Index: linux-2.6.16-rt23.spin_mutex/kernel/irq/manage.c
=====
--- linux-2.6.16-rt23.spin_mutex.orig/kernel/irq/manage.c
+++ linux-2.6.16-rt23.spin_mutex/kernel/irq/manage.c
@@ -206,6 +206,153 @@ int can_request_irq(unsigned int irq, un
return !action;
}

+
+#ifdef CONFIG_CHANGE_IRQ_CONTEXT
+int make_action_nodelay(int irq, struct irqaction *act)
+{
+ unsigned long flags;
+
+ if(!(act->flags & SA_MUST_THREAD)) {
+ return 0;
+ }
+ }
```

Remove the brackets.

Also, let me get this straight. If the action does not have SA_MUST_THREAD, we return? Or does it mean that if SA_MUST_THREAD is not set, then SA_NODELAY must already be set? If that is the case, then why are we not checking for SA_NODELAY here?

If SA_MUST_THREAD is not set, there is no blocking inside the action handler and therefore it is ok to do this in hardirq aka "nodelay".

Notice SA_MUST_THREAD is set on all handlers in without SA_NODELAY under PREEMPT_RT.

Re: [patch 2/5] [PREEMPT_RT] Changing interrupt handlers from running in thread to hardirq and back runtime

Re: [patch 2/5] [PREEMPT_RT] Changing interrupt handlers from running in thread to hardirq and back runtime.

```
+
+ if( act->change_context ) {
+ int ret =
+ act->change_context(irq, act->dev_id, IRQ_TO_HARDIRQ);
+ if(ret) {
+ printk(KERN_ERR "Failed to change irq handler "
+ "for dev %s on IRQ%d to hardirq "
+ "context (ret: %d)\n", act->name, irq, ret);
+ return ret;
+ }
+ spin_lock_irqsave(&irq_desc[irq].lock, flags);
+ act->flags &= ~SA_MUST_THREAD;
```

Both flags are zero here (see below about the WARN_ON)

The WARN_ON is a warning on both flags set (or supposed to be...:-)
It doesn't make sense to have SA_MUST_THREAD and SA_NODELAY both set.

```
+ act->flags |= SA_NODELAY;
+ spin_unlock_irqrestore(&irq_desc[irq].lock, flags);
+ return 0;
+ }
+ else {
+ printk(KERN_ERR "Irq handler "
+ "for dev %s on IRQ%d can not be changed"
+ " to hardirq context\n", act->name, irq);
+ return -ENOSYS;
+ }
+
+
+
+static int __make_irq_threaded(int irq, struct irq_desc *desc);
+
+int make_irq_nodelay(int irq, struct irq_desc *desc)
+{
+ int ret = 0;
+ struct irqaction *act;
+ unsigned long flags;
+
+ rcu_read_lock();
+ for(act = desc->action; act; act = act->next) {
+ WARN_ON(((~act->flags) & (SA_MUST_THREAD|SA_NODELAY))
+ == 0);
```

Re: [patch 2/5] [PREEMPT_RT] Changing interrupt handlers from running in thread to hardirq and back runtime

Re: [patch 2/5] [PREEMPT_RT] Changing interrupt handlers from running in thread to hardirq and back runtime.

This WARN_ON is not protected by the descriptor lock, so if this function is run on two CPUs at the same time, then the act->flags can have both of these zero by the above code.

Both be set :-) But bug noticed.

```
+ if(act->flags & SA_MUST_THREAD) {
+ ret = make_action_nodelay(irq, act);
+ if(ret) {
+ printk(KERN_ERR "Could not make irq %d "
+ "nodelay (errno %d)\n",
+ irq, ret);
```

We printk on failure from above, and then printk again here?

Well might be too verbose.

```
+ goto failed;
+ }
+ }
+ }
+ spin_lock_irqsave(&desc->lock, flags);
+ desc->status |= IRQ_NODELAY;
+ spin_unlock_irqrestore(&desc->lock, flags);
+ rcu_read_unlock();
+
+ return 0;
+ failed:
+ __make_irq_threaded(irq, desc);
+ rcu_read_unlock();
+ return ret;
+ }
+
+int action_may_thread(int irq, struct irqaction *act)
+{
+ if(!act->change_context)
+ return !(act->flags & SA_NODELAY);
+
+ return act->change_context(irq, act->dev_id, IRQ_CAN_THREAD) == 0;
```

Re: [patch 2/5] [PREEMPT_RT] Changing interrupt handlers from running in thread to hardirq and back runtime

Re: [patch 2/5] [PREEMPT_RT] Changing interrupt handlers from running in thread to hardirq and back runtime.

This `IRQ_CAN_THREAD` just looks out of place of the other two. It feels very "hacky" to check if it can thread. But what do I know?

The problem is that the step to make a handler be threaded has to be split in two: First ask, then remove `IRQ_NODELAY`, then change the locks to mutexes. So the driver need to be involved twice. I could skip this, but I put it in for generallity.

```
+}
+
+
+static int __make_irq_threaded(int irq, struct irq_desc *desc)
+{
+ struct irqaction *act;
+
+ for(act = desc->action; act; act = act->next) {
+ WARN_ON(((~act->flags) & (SA_MUST_THREAD|SA_NODELAY))
+ == 0);
+ if(!action_may_thread(irq, act)) {
+ return -EINVAL;
+ }
+ }
+
+ if (start_irq_thread(irq, desc))
+ return -ENOMEM;
```

I know that currently `start_irq_thread` can only return `-ENOMEM` on failure, but it may be more robust to capture the return code and return that anyway.

nod

This was copied from the original code, so the same thing is elsewhere, too.

```
+
+ spin_lock_irq(&desc->lock);
+ desc->status &= ~IRQ_NODELAY;
+ spin_unlock_irq(&desc->lock);
+
+ /* We can't make irq handlers change their context before we
+ are sure no CPU is running them in hard irq context */
+ synchronize_irq(irq);
```

Re: [patch 2/5] [PREEMPT_RT] Changing interrupt handlers from running in thread to hardirq and back runtime

Re: [patch 2/5] [PREEMPT_RT] Changing interrupt handlers from running in thread to hardirq and back runtime.

```
+
+ for(act = desc->action; act; act = act->next) {
+ WARN_ON(((~act->flags) & (SA_MUST_THREAD|SA_NODELAY))
== 0);
+ if(act->change_context) {
+ /* This callback can't fail */
```

Will all device drivers know that the callback can't fail?

Well, it is in the documentation.

```
+ act->change_context(irq, act->dev_id, IRQ_TO_THREADED);
+ spin_lock_irq(&desc->lock);
+ act->flags &=~SA_NODELAY;
+ act->flags |= SA_MUST_THREAD;
+ spin_unlock_irq(&desc->lock);
+ }
+ }
+
+ return 0;
+}
```

[snipped the rest]

— Steve

—
To unsubscribe from this list: send the line "unsubscribe linux-kernel" in
the body of a message to majordomo@xxxxxxxxxxxxxxxxx
More majordomo info at <http://vger.kernel.org/majordomo-info.html>
Please read the FAQ at <http://www.tux.org/lkml/>

Re: [patch 2/5] [PREEMPT_RT] Changing interrupt handlers from running in thread to hardirq and back runtime