Using RCU to Protect Dynamic NMI Handlers

Although RCU is usually used to protect read-mostly data structures, it is possible to use RCU to provide dynamic non-maskable interrupt handlers, as well as dynamic irq handlers. This document describes how to do this, drawing loosely from Zwane Mwaikambo's NMI-timer work in "arch/i386/oprofile/nmi_timer_int.c" and in "arch/i386/kernel/traps.c".

The relevant pieces of code are listed below, each followed by a brief explanation.

```
static int dummy_nmi_callback(struct pt_regs *regs, int cpu)
{
        return 0;
}
```

The dummy_nmi_callback() function is a "dummy" NMI handler that does nothing, but returns zero, thus saying that it did nothing, allowing the NMI handler to take the default machine-specific action.

```
static nmi_callback_t nmi_callback = dummy_nmi_callback;
```

This nmi_callback variable is a global function pointer to the current NMI handler.

```
void do_nmi(struct pt_regs * regs, long error_code)
{
        int cpu;

        nmi_enter();

        cpu = smp_processor_id();
        ++nmi_count(cpu);

        if (!rcu_dereference_sched(nmi_callback)(regs, cpu))
                default_do_nmi(regs);

        nmi_exit();
}
```

The do_nmi() function processes each NMI. It first disables preemption in the same way that a hardware irq would, then increments the per-CPU count of NMIs. It then invokes the NMI handler stored in the nmi_callback function pointer. If this handler returns zero, do_nmi() invokes the default_do_nmi() function to handle a machine-specific NMI. Finally, preemption is restored.

In theory, rcu_dereference_sched() is not needed, since this code runs only on i386, which in theory does not need rcu_dereference_sched() anyway. However, in practice it is a good documentation aid, particularly for anyone attempting to do something similar on Alpha or on systems with aggressive optimizing compilers.

Quick Quiz: Why might the rcu_dereference_sched() be necessary on Alpha,

given that the code referenced by the pointer is read-only?

Back to the discussion of NMI and RCU...

```
void set_nmi_callback(nmi_callback_t callback)
{
        rcu_assign_pointer(nmi_callback, callback);
}
```

The set_nmi_callback() function registers an NMI handler. Note that any data that is to be used by the callback must be initialized up -before- the call to set_nmi_callback(). On architectures that do not order writes, the rcu_assign_pointer() ensures that the NMI handler sees the initialized values.

```
void unset_nmi_callback(void)
{
        rcu_assign_pointer(nmi_callback, dummy_nmi_callback);
}
```

This function unregisters an NMI handler, restoring the original dummy_nmi_handler(). However, there may well be an NMI handler currently executing on some other CPU. We therefore cannot free up any data structures used by the old NMI handler until execution of it completes on all other CPUs.

One way to accomplish this is via synchronize_sched(), perhaps as follows:

```
unset_nmi_callback();
synchronize_sched();
kfree(my_nmi_data);
```

This works because synchronize_sched() blocks until all CPUs complete any preemption-disabled segments of code that they were executing. Since NMI handlers disable preemption, synchronize_sched() is guaranteed not to return until all ongoing NMI handlers exit. It is therefore safe to free up the handler's data as soon as synchronize_sched() returns.

Important note: for this to work, the architecture in question must invoke irq_enter() and irq_exit() on NMI entry and exit, respectively.

Answer to Quick Quiz

Why might the rcu_dereference_sched() be necessary on Alpha, given that the code referenced by the pointer is read-only?

Answer: The caller to set_nmi_callback() might well have initialized some data that is to be used by the new NMI handler. In this case, the rcu_dereference_sched() would be needed, because otherwise a CPU that received an NMI just after the new handler was set might see the pointer to the new NMI handler, but the old pre-initialized version of the handler's data.

This same sad story can happen on other CPUs when using a compiler with aggressive pointer-value speculation optimizations.

More important, the rcu_dereference_sched() makes it clear to someone reading the code that the pointer is being protected by RCU-sched.