SMP Critical Sections

Single CPU Critical Sections

OS Interfaces

Before we talk about SMP Critical Sections let's first review the internal OS interfaces avaiable and what they do in the single CPU case:

- `up_irq_save()` (and its companion, `up_irq_restore()`). These simple interfaces just enable and disable interrupts globally. This is the simplest way to establish a critical section in the single CPU case. It does have side-effects to real-time behavior as discussed elsewhere [http://www.nuttx.org/doku.php?id=wiki:nxinternal:disable_latency].
- `up_irq_save()` should *never* be called directly, however. Instead, the wrapper *macros* `enter_critical_section()` (and its companion `leave_critical_section()`) or `spin_lock_irqsave()` (and `spin_unlock_irqrestore()`) should be used. In the single CPU case, these macros are defined to be simply `up_irq_save()` (or `up_irq_save()`). Rather than being called directly, they should always be called indirectly through these macros so that the code will function in the SMP environment as well.
- Finally, there is `sched_lock()` (and `sched_unlock()`) that disable (and enable) pre-emption. That is, `sched_lock()` will *lock* your kernel thread in place and prevent other tasks from running. Interrupts are still enabled, but other tasks cannot run.

Using sched_lock() for Critical Sections -- DON'T

In the single CPU case, `sched_lock()` can do a pretty good job of establishing a critical section too. After all, if no other tasks can run on the single CPU, then that task has pretty much exclusive access to all resources (provided that those resources are not shared with interrupt handlers). However, `sched_lock()` must never be used to establish a critical section because it does not work the same way in the SMP case. In the SMP case, locking the scheduer does not provide any kind of exclusive access to resources. Tasks running on other CPUs are still free to do whatever they wish.

SMP Critical Sections

up_irq_save() and up_irq_restore()

As mentioned, `up_irq_save()` and `up_irq_restore()` should never be called directly. That is because the behavior is different in multiple CPU systems. In the multiple CPU case, these functions only enable (or disable) interrupts on the *local* CPU. They have no effect on interrupts in the other CPUs and hence really accomplish very little. Certainly they do not provide a critical section in any sense.

enter_critical_section() and leave_critical_section()

spinlocks

In order to establish a critical section, we also need to employ *spinlocks*. Spins locks are simply loops that execute in one processor. If processor A sets spinlock x, then processor B would have to wait for the spinlock like:

```
  while (test_and_set(x))
    {
    }
```

Where *test and set* is an atomic operation that sets the value of a memory location but also returns its previous value. Here we are talking about atomic in terms of memory bus operations: The testing and setting of the memory location must be atomic with respect to other bus operations. Special hardware support of some kind is necessary to implement `test_and_set()` logic.

When Task A released the lock x, Task B will successfully take the spinlock and continue.

Implementation

Without going into the details of the implementation of `enter_critical_section()` suffice it to say that it (1) disables interrupts on the local CPU and (2) uses spinlocks to assure exclusive access to a code sequence across all CPUs.

NOTE that a critical section is indeed created: While within the critical section, the code does have exclusive access to the resource being protected. However the behavior is really very different:

- In the single CPU case, disable interrupts stops all possible activity from any other task. The single CPU becomes single threaded and un-interruptible.
- In the SMP case, tasks continue to run on other CPUs. It is only when those other tasks attempt to enter a code sequence protected by the critical section that those tasks on other CPUs will be stopped. They will be stopped waiting on a spinlock.

spin_lock_irqsave() and spin_unlock_irqrestore()

Generic Interrupt Controller (GIC)

ARM provides a special, optional sub-system called MPCore that provides multi-core support. One MPCore component is the *Generic Interrupt Controller* or GIC. The GIC supports 16 inter-processor interrupts and is a key component for implementing SMP on those platforms. The are called *Software Generated Interrupts* or SGIs.

One odd behavior of the GIC is that the SGIs cannot be disabled (at least not using the standard ARM global interrupt disable logic). So disabling local interrupts does not prevent these GIC interrupts.

This causes numerous complexities and significant overhead in establishing a critical section.

NOTE: It might be possible to disable these inter-processor interrupts priority levels. From the TODO list: "Masayuki Ishakawa has suggested the use of the GICv2 ICCMPR register to control SGI interrupts. This register (much like the ARMv7-M BASEPRI register) can be used to mask interrupts by interrupt priority. Since SGIs may be assigned priorities the ICCMPR should be able to block execution of SGIs as well. …"

ARMv7-M NVIC

The GIC is available in all recent ARM architectures. However, most embedded ARM7-M multi-core CPUs just incorporate the inter-processor interrupts as a normal interrupt that is mask-able via the NVIC (each CPU will have its own NVIC).

This means in those cases, the critical section logic can be greatly simplified.

Implementation

For the case of the GIC with no support for disabling interrupts, `spin_lock_irqsave()` and `spin_unlock_irqstore()` are equivalent to `enter_critical_section()` and `leave_critical_section()`. In is only in the case where inter-processor interrupts can be disabled that there is a difference.

In that case, `spin_unlock_irqsave()` will disable local interrupts and take a spinlock. This is really very simple and efficient implementation of a critical section.

There are two important things to note, however:

1. The logic within this critical section must never suspend! For example, if code were to call `spin_unlock_irqsave()` then `sleep()`, then the sleep would occur with the spinlock in the lock

state and the whole system could be blocked. Rather, `spin_unlock_irqsave()` can only be used with straight line code.
2. This is a different critical section than the one established via `enter_critical_section()`. Taking one critical section, does not prevent logic on another CPU from taking the other critical section and the result is that you make not have the protection that you think you have.

sched_lock() and sched_unlock()

Other than some details, the SMP `sched_lock()` works much like it does in the single CPU case. Here are the caveats:

- As in the single CPU case, the case that calls `sched_lock()` is locked in place and cannot be suspected.
- However, tasks will continue to run on other CPUs so `sched_lock()` cannot be used as a critical section.
- Tasks on other CPUs are also locked in place. However, they may opt to suspend themselves at any time (say, via `sleep()`). In that case, only the CPU's IDLE task will be permitted to run.
