Improvements to Interrupt Thread Scheduling Latency#

The QNX microkernel only allows one thread to 'own' the kernel at a time. If a higher priority thread becomes scheduled to run because of an interrupt, then the currently executing kernel call will be preempted, and the new thread will be put into execution.

If the kernel call has locked the kernel, only voluntary preemption is possible. The kernel call can check the value of queued_event_priority to see if a higher priority thread has been made ready as a result of an interrupt handler.

This works well on a uniprocessor system, but on an SMP system, a thread executing on another processor may make a kernel call and try to acquire the inkernel ownership. Prior to 6.4.1 this would involve executing a busy-wait loop.

An example problem scenario is a low priority thread could be performing a long lasting kernel call, and on another processor a higher priority thread is attempting to make a short kernel call, and would have to wait for the duration of the kernel call.

Another example is if two multiple processors attempt to make a kernel call at the same time, then the order of inkernel acquisition was dependent on the architecture of the memory bus.

6.4.1 Improvements#

To address these problems, the kernel acquisition code was changed to attempt the inkernel acquisition once, and if that fails to call a new function, inkernel_wait(). This places the processor into an array of ordered lists (similar to the dispatch array), where each list corresponds to a priority. The lists are sorted in FIFO order.

The core will then resume spinning attempting to acquire the kernel, but it is now moderated by the list - it can only acquire the kernel if it is the highest priority thread waiting for the kernel, and if it is the head of it's priority queue.

Here is the pseudo code for the inkernel_wait function...

```
inkernel_wait() {

   for( ;; ) {

      calculate our effective priority

      if (already waiting and our priority has changed)
         remove ourselves position in the waitin_cpus

      place ourselves in the waiting_cpus list
      update the bitmask of waiting cpus

      update max_ker_waiting_prio to be the priority of the highest thread

      Bailout 1: If a thread on another core is also trying to
      enter the kernel, and it has a higher prio than us, then
      defer to it, then continue the loop

      Bailout 2: If we are not the head of our priority queue - ie others
      have been waiting longer than us, then continue the loop

      Bailout 3: If omeone else already has the kernel lock, then continue the loop

      Attempt to acquire ownership of inkernel
```

```
        If we succeeded, then break out of loop
    }

    Remove ourselves from the waiting_cpus list and update max_ker_waiting_prio
}
```

Now the kernel preemption checks are based on both the queued_event_priority AND the max_ker_waiting_prio variables, so that if there is a higher priority thread wanting the kernel on another CPU, the kernel can tell.

Supported Architectures#

For 6.4.1 only the x86 implementation is planned.

Design and Code Review#

Design review was performed at the kernel tech talk on the 24th of March, 2009, which was attended by the kernel group, including Brian Stecher, Doug Bailey, Attila Danko and Steve Bergwerff. No objections were raised to the design

The code review discussion can be found here... http://community.qnx.com/sf/go/post26767