Using Interrupt Threads to Prioritize Interrupts

Aren’t Interrupts the Highest Priority Already?

Mike Anderson
Chief Scientist
The PTR Group, Inc.
mailto: mike@theptrgroup.com
http://www.theptrgroup.com

What We Will Talk About

❖ What is latency?
❖ Sources of latency in real-time systems
❖ Misconceptions about interrupt service routines
❖ Executing interrupt code in a thread context
❖ Interrupt threads in Linux
❖ Some notional performance comparisons
❖ Summary
A Definition of Latency

Latency can best be described as the difference in time between when an event is signaled and when code starts to run.

Operating systems have:
  › Scheduling latency
  › Interrupt latency
  › And more...

Because we deal with the real world, we must deal with latency.
  › The real world is not a very deterministic place

Scheduling Latency

Scheduling latency is the amount of time between when a high-priority thread becomes ready to run and when it gets the CPU.

Affected by:
  › Disabling the scheduler
    • E.g., the BKL in Linux or taskLock() in VxWorks™
  › Non-preemptible system calls
Interrupt Latency

- The amount of time between when an interrupt is signaled and when the ISR begins to execute
- Affected by:
  - Long-duration ISRs
  - Disabling interrupts
  - The order of interrupt arrival

Taxonomy

- Deterministic execution
  - This means that code takes the same amount of time to run every time
    - The holy grail of real-time systems
- Real-time computing
  - Computing with a deadline
- Soft real time
  - Deadlines are squishy
    - Executing after the deadline has diminishing value
- Hard real time
  - If you miss the deadline, people get hurt or data is lost permanently
Real-time isn’t Fair

✦ Embedded RTOS developers know that real-time applications are decidedly unfair
  ▸ Time slicing may or may not exist in your RTOS
✦ In fact, many RTOSes don’t support round-robin scheduling very well
  ▸ Preemptive, priority-based is the scheduler of choice
    • That’s SCHED_FIFO to us Linux folks
✦ This unfairness requires a different mindset from traditional desktop development
  ▸ Can take some getting used to

Preemption in the O/S Kernel

✦ Ideally, an embedded O/S kernel should be fully preemptible
  ▸ Being fully preemptible enables the most responsiveness to high-priority code
    • Unfortunately, it may also reduce throughput
✦ Not all kernels are fully preemptible
  ▸ Early Linux was a good example of this
✦ Nearly all kernels have some regions of non-preemptibility
  ▸ Semaphore operations, memory allocation, ISR dispatch, etc.
  ▸ The number and duration of these regions will impact responsiveness
Kernel Preemption w/ Low-Latency Desktop

MP3 without Preemption  MP3 with Preemption

Selecting Preemption Models in Linux

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>General setup</td>
<td>Configure standard kernel features for small systems</td>
</tr>
<tr>
<td>Linux Trace Toolkit</td>
<td>Enable the block layer</td>
</tr>
<tr>
<td>I/O schedulers</td>
<td>System Type</td>
</tr>
<tr>
<td>OMAP implementations</td>
<td>Bus support</td>
</tr>
<tr>
<td>PCM.Card (PCMIA/CardBus) support</td>
<td>Boot options</td>
</tr>
<tr>
<td>CPU Frequency scaling</td>
<td>CPU idle PM support</td>
</tr>
<tr>
<td>Floating point emulation</td>
<td>Power management options</td>
</tr>
<tr>
<td>User-space binary formats</td>
<td>OMAP power management options</td>
</tr>
<tr>
<td>Networking</td>
<td>Device Drivers</td>
</tr>
<tr>
<td>Generic Driver Options</td>
<td>Memory Technology Device (MTD) support</td>
</tr>
<tr>
<td>Memory Technology Device (MTD) support</td>
<td>ARM/Non-Flash chip drivers</td>
</tr>
<tr>
<td>ARM/Non-Flash chip drivers</td>
<td>Self-contained MTD device drivers</td>
</tr>
<tr>
<td>Self-contained MTD device drivers</td>
<td>DMA/Interrupt Device Support</td>
</tr>
<tr>
<td>DMA/Interrupt Device Support</td>
<td>UDF - Unsorted block images</td>
</tr>
<tr>
<td>UDF - Unsorted block images</td>
<td>Parallel port support</td>
</tr>
<tr>
<td>Parallel port support</td>
<td>Back devices</td>
</tr>
<tr>
<td>Back devices</td>
<td>BMC devices</td>
</tr>
</tbody>
</table>

No Forced Preemption (Server) (PREEMPT_NONE)

This is the traditional Linux preemption model geared towards throughput. It will still provide good latencies most of the time but there are no guarantees and occasional long delays are possible.

Select this option if you are building a kernel for a server or scientific/computation system, or if you want to minimize the new processing power of the kernel, irrespective of scheduling latencies.
Preemption Latency is Key

- So, why the emphasis on task preemption latency?
  - If we run our ISRs in threads, we need to know how long before they can run
- The more responsive the kernel, the more responsive our interrupt threads
- Long duration, non-preemptible system calls will kill our performance with the techniques we’ll discuss

Prioritizing Interrupts

- For most of us, ISRs represent the highest priority entity in our system
- The venerable VMEbus supported an interrupt hierarchy
  - Int 5 could preempt Int 4 but not Int 6
- Unfortunately, PCI bus doesn’t support interrupt priorities
  - Any interrupt can preempt any other interrupt
  - Interrupt sharing can make interrupt chains incredibly long running
- We’d like to be able to prioritize PCI interrupts as well
Notional Linux IRQ Action Table

Breaking Training

We’ve been trained to think that interrupt code must be:

- Fast
- Atomic
- Run in a special context

But, what processor instructions *must* be run in interrupt context?

- Return from interrupt
  - E.g., PPC RFI or x86 IRET
- That’s about it

OK, what about fast and atomic?
How Fast is Fast Enough?

- Well, it depends…
  - Do we have a buffer that will be overrun?
  - When does the hardware interrupt get re-enabled?
- Examples such as the Linux kernel NAPI interface shows us that we can reduce the number of interrupts and still have excellent service
  - Buffering may be automatic and in hardware
- If we have to re-arm the interrupt in our ISR, then it’s likely that the re-arm can wait until we get to it
  - Will data be lost? Is it important?

OK, How about Atomic?

- Many O/Ses support the concept of nested interrupts
  - E.g., interrupts masked at the PIC rather than at the CPU itself
  - Our interrupt stack must handle worst case nesting
- By their nature, nested interrupts are not atomic
  - I could be in the middle of an ISR and get interrupted by another interrupt
- It’s likely important to prioritize interrupts
  - We may want highest priority interrupt to run to completion
  - Especially true for mission-critical systems
ISR Latency Sources

The most significant ISR latencies come from two sources

- Disabling interrupts in driver or user code
  - E.g., `local_irq_disable()/local_irq_enable()` in Linux
- Performing non-deterministic operations in the ISR itself
  - E.g., copying packets from network hardware during the ISR

A common technique is to separate the code which must be done immediately from the code that is non-deterministic

- Known as top-half/bottom-half approach

Top vs. Bottom Half

The goal of this approach is to make the top half deterministic

- Maybe just acknowledge the IRQ and then schedule post-interrupt work
  - E.g., A tasklet in Linux
- The bottom half runs in a different context
- The interrupts are re-enabled
- Lengthy copy operations are moved here
- Rearming the IRQ is the last thing you do

The bottom half is usually dispatched as a software ISR

- Little or no ability to prioritize
Interrupt Latency Reduction

- We’ve learned to use bottom halves to reduce interrupt latency
  - Lengthy copy operations can be moved to SoftIRQ/tasklet/work queue to re-enable interrupts while the copy proceeds
- Work queues are kernel threads
  - They’re scheduled, have priorities and can sleep
- The ISR top half can be a single schedule_work() call
  - This makes the top half deterministic

Scheduling Work

- The Linux scheduler is O(1)
  - Deterministic dispatch time
- This means that the work queue will be scheduled in constant time
- Since the work queue is a thread, it can run as long as needed (SCHED_FIFO)
  - Highest priority wins with the scheduler
- This means we can use R–T priorities to prioritize execution of bottom half
  - This is something we didn’t have with tasklets/softIRQs
R–T Patch to the Rescue

- What the R–T patch does is to institutionalize the work queue idea
  - All hardIRQs and softIRQs execute in high-priority kernel threads
- Highest priority wins
- Threaded hard and soft IRQs can be disabled via kernel command line or in /proc
  - hardirq-preempt=0/1
  - /proc/sys/kernel/hardirq_preemption
  - Similar options for softIRQs

Threads are Created Automatically

- You don’t have to do anything special to run your code in a thread
  - request_irq() call creates the thread and includes your function
    ```c
    if (!(new->flags & IRQF_NODELAY))
        if (start_irq_thread(irq, desc))
            return -ENOMEM;
    ```
- This code will pass your ISR to the start_irq_thread function
  - Creates a kernel thread that calls your ISR code
The start_irq_thread Call

```c
static int start_irq_thread(int irq, struct irq_desc *desc)
{
    if (!desc->thread || !ok_to_create_irq_threads)
        return 0;

    desc->thread = kthread_create(do_irqd, desc, "IRQ-%d", irq);
    if (!desc->thread) {
        printk(KERN_ERR "irqd: could not create IRQ thread %d!\n", irq);
        return -ENOMEM;
    }

    /*
     * An interrupt may have come in before the thread pointer was
     * stored in desc->thread; make sure the thread gets woken up in
     * such a case:
     */
    smp_mb();
    wake_up_process(desc->thread);

    return 0;
}
```

Prioritizing Interrupts w/ Interrupt Threads

- We associate each ISR with a unique thread
  - Each thread has its own priority
  - Threads of the same priority will run back-to-back in the order they were scheduled

- By keeping the ISR short (just schedule the thread), we make ISR top halves deterministic
  - The deterministic scheduler then schedules the highest priority thread
    - Hardware IRQ prioritization is a side effect

- This means that interrupt thread dispatch is deterministic
  - What you do in the thread doesn’t have to be

- Only another ISR or higher priority thread will preempt you
  - This is what we want anyway
Reduction of Jitter due to Latency

Here is an example of latency reduction due to the use of interrupt threads

Difference between the green and blue lines is the use of ISR threads

View of Threaded IRQs in Linux

With the RT patch set enabled, the hard/softIRQs are automatically run in kernel threads

- Kernel threads use the kernel’s API and share the address space with drivers, the kernel etc.

<table>
<thead>
<tr>
<th>PID</th>
<th>TID</th>
<th>CLS</th>
<th>RTPRIO</th>
<th>NI</th>
<th>PRI</th>
<th>PSR</th>
<th>CPU</th>
<th>STAT</th>
<th>COMMAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>TS</td>
<td>-</td>
<td>0</td>
<td>19</td>
<td>0</td>
<td>0.0</td>
<td>S</td>
<td>init</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>TS</td>
<td>-</td>
<td>5</td>
<td>24</td>
<td>0</td>
<td>0.0</td>
<td>S</td>
<td>kthread</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>FF</td>
<td>90</td>
<td>139</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
<td>S</td>
<td>migration/0</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>FF</td>
<td>90</td>
<td>139</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
<td>S</td>
<td>posix_cpu_timer</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>FF</td>
<td>50</td>
<td>90</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
<td>S</td>
<td>softirq-high/0</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>FF</td>
<td>50</td>
<td>90</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
<td>S</td>
<td>softirq-timer/0</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>FF</td>
<td>50</td>
<td>90</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
<td>S</td>
<td>softirq-net-tx/</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>FF</td>
<td>50</td>
<td>90</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
<td>S</td>
<td>softirq-net-rx/</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>FF</td>
<td>50</td>
<td>90</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
<td>S</td>
<td>softirq-block/0</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>FF</td>
<td>50</td>
<td>90</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
<td>S</td>
<td>softirq-tasklet</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>FF</td>
<td>50</td>
<td>90</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
<td>S</td>
<td>softirq-sched/0</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td>FF</td>
<td>50</td>
<td>90</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
<td>S</td>
<td>softirq-hrtimer</td>
</tr>
<tr>
<td>13</td>
<td>13</td>
<td>FF</td>
<td>50</td>
<td>90</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
<td>S</td>
<td>softirq-rcu/0</td>
</tr>
<tr>
<td>56</td>
<td>56</td>
<td>FF</td>
<td>50</td>
<td>90</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
<td>S</td>
<td>IRQ-9</td>
</tr>
<tr>
<td>884</td>
<td>884</td>
<td>FF</td>
<td>50</td>
<td>90</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
<td>S</td>
<td>IRQ-8</td>
</tr>
<tr>
<td>922</td>
<td>922</td>
<td>FF</td>
<td>50</td>
<td>90</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
<td>S</td>
<td>IRQ-12</td>
</tr>
<tr>
<td>923</td>
<td>923</td>
<td>FF</td>
<td>50</td>
<td>90</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
<td>S</td>
<td>IRQ-1</td>
</tr>
</tbody>
</table>
Not Every ISR Should be Threaded

- You do not have to thread all your ISRs
  - Just because you can doesn’t mean you should
- There are some classes of ISRs where this approach doesn’t make sense
  - Timer ISRs
  - ISRs that are already deterministic like receiving on a serial port
    - The overhead of scheduling exceeds their nominal run time
- Just try to make sure that everything is as deterministic as possible
  - Make sure that you measure it afterwards to verify you actually improved responsiveness

Deterministic may not be Faster

- Because of the issues of preemption and the overhead of running the scheduler, interrupt threads may not be faster than the old approach
  - It’s deterministic, but not faster
- A good trade for some applications
  - A bad one for others
- That’s why you need to use interrupt threads only where they make sense
- Interrupt reduction techniques may also be important
  - Don’t immediately re-enable the IRQ in the bottom half
    - Poll the device instead to avoid overhead of servicing IRQ and rescheduling
Summary

🌟 Real-time means being fast enough
  - Determinism is nice to have when you can get it
    • Some applications, like audio, require it

🌟 The use of interrupt threads enables developers to prioritize interrupts and make interrupt servicing more deterministic
  - Jitter goes way down
  - May require some system redesign to take full advantage of threading

🌟 Use interrupt threads judiciously
  - Not every ISR needs this approach