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

Source: graphpaper.com

- 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

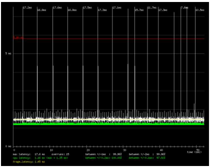
Real-time isn't Fair

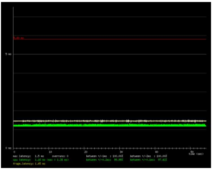
- Embedded RTOS developers know that real-time applications are decidedly unfair
  - Time slicing may or may not exist in your RTOS
- In fact, many RTOSes don't support round-robin scheduling very well
  - Preemptive, priority-based is the scheduler of choice
    - That's SCHED_FIFO to us Linux folks
- 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

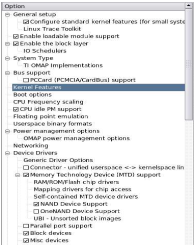


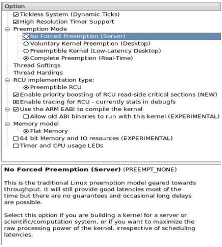


Source: linuxjournal.com

MP3 without Preemption | MP3 with Preemption

Selecting Preemption Models in Linux





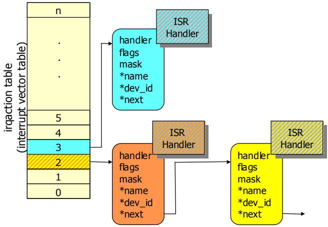
Preemption Latency is Key

- 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

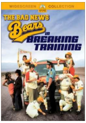
Notional Linux IRQ Action Table



Shared Interrupt Chain

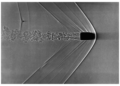
Breaking Training

- 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?



Source: sportonvideo.com

How Fast is Fast Enough?



Source: nasa.gov

- 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?

Source: deskpicture.com

- 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

Source: jenklairkids.com

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

Yang's Nail Salon
Work Schedule – Week of June 15 – 20

Mon 9 – 5	Tues 9 – 5	Wed 9 – 5	Thurs 10 – 6	Fri 12 – 8	Sat 12 – 8
Manicurists					
* Kim	Kim	* Kim	* Kim	Kim	
Barb	* Barb	Barb	Barb		* Barb
Lin				Lin	Lin
	Jon	* Jon	Jon	Jon	Jon
	* Lisa	Lisa	* Lisa	Lisa	* Lisa
	Huy	Huy	Huy	* Huy	Huy
Pedicurists					
	Pat	Pat			* Pat
			Sara	* Sara	Sara
			Mary	Mary	
Receptionists					
Sofia	Sofia	Sofia	Sofia		*Sofia
				* Ann	Ann
				Mary	
* Early lunch 11:30 – 12:00 (Others: 12:00 – 12:30)					

Source: johnmh.com

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

Source: incredimazing.com

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

```
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

```
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;
}
```

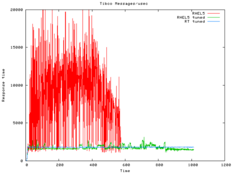

Prioritizing Interrupts w/ Interrupt Threads

- 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

Source: wordpress.com

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



Source: redhat.com

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.

```
  PID   TID CLS RTPRIO  NI PRI PSR %CPU STAT COMMAND
    1     1 TS       -   0  19   0  0.0 Ss   init
    2     2 TS       -  -5  24   0  0.0 S<   kthreadd
    3     3 FF      99   - 139   0  0.0 S<   migration/0
    4     4 FF      99   - 139   0  0.0 S<   posix_cpu_timer
    5     5 FF      50   -  90   0  0.0 S<   softirq-high/0
    6     6 FF      50   -  90   0  0.5 S<   softirq-timer/0
    7     7 FF      50   -  90   0  0.0 S<   softirq-net-tx/
    8     8 FF      50   -  90   0  0.0 S<   softirq-net-rx/
    9     9 FF      50   -  90   0  0.0 S<   softirq-block/0
   10    10 FF      50   -  90   0  0.0 S<   softirq-tasklet
   11    11 FF      50   -  90   0  0.0 S<   softirq-sched/0
   12    12 FF      50   -  90   0  0.0 S<   softirq-hrtimer
   13    13 FF      50   -  90   0  0.0 S<   softirq-rcu/0
   56    56 FF      50   -  90   0  0.0 S<   IRQ-9
  884   884 FF      50   -  90   0  0.0 S<   IRQ-8
  922   922 FF      50   -  90   0  0.0 S<   IRQ-12
  923   923 FF      50   -  90   0  0.0 S<   IRQ-1
```

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

Source: germes-online.com

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