The Timer Resolution Latency

Luca Abeni
luca.abeni@unitn.it

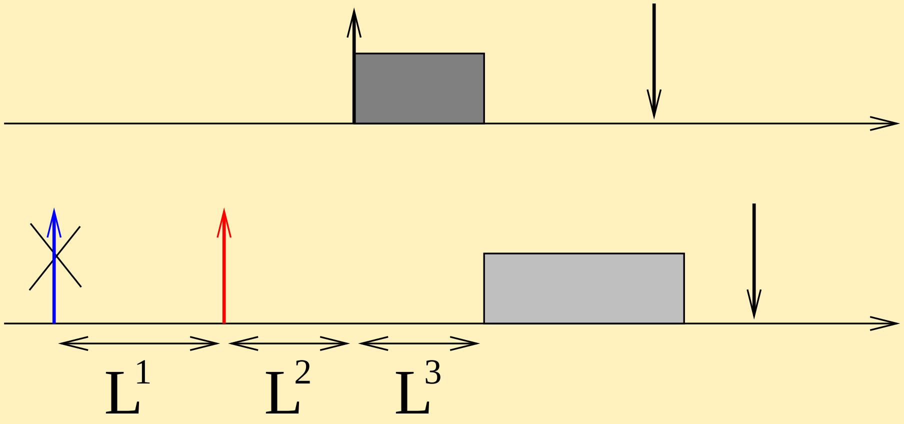
November 24, 2014

Latency

- Latency: measure of the difference between the theoretical and actual schedule
    - Task τ expects to be scheduled at time t ...
    - ... but is scheduled at time t'
    - $\Rightarrow$ Latency $L = t' - t$
- The latency L can be modelled as a blocking time $\Rightarrow$ affects the guarantee test
- If L is too high, only few task sets result to be schedulable
    - The latency must be *bounded*: $\exists L^{max} : L < L^{max}$
    - The latency bound L^{max} cannot be too high

Sources of Latency

- A task τ_i is a stream of jobs $J_{i,j}$ arriving at time $r_{i,j}$
- Job $J_{i,j}$ is scheduled at time $t' > r_{i,j}$
  - $t' - r_{i,j}$ is given by the sum of various components:
    1. $J_{i,j}$'s arrival is signalled at time $r_{i,j} + L^1$
    2. Such event is served at time $r_{i,j} + L^1 + L^2$
    3. $J_{i,j}$ is actually scheduled at $r_{i,j} + L^1 + L^2 + L^3$



Analysis of the Various Sources

- $L = L^1 + L^2 + L^3$
- L^3 is the *scheduler latency*
    - Interference from higher priority tasks
    - Already accounted by the guarantee tests → let's not consider it
- L^2 is the *non-preemptable section latency*, called L^{np}
    - Due to non-preemptable sections in the kernel, which delays the response to hardware interrupts
    - It is composed by various parts: *interrupt disabling*, *bottom halves delaying*, $\ldots$
- L^1 is due to the delayed interrupt generation

Interrupt Generation Latency

- Hardware interrupts are generated by external devices
- Sometimes, a device must generate an interrupt at time t ...
- ... but actually generates it at time $t' = t + L^{int}$
- L^{int} is the *Interrupt Generation Latency*
    - It is due to hardware issues
    - It is *generally* small compared to L^{np}
    - Exception: if the device is a timer device, the interrupt generation latency can be quite high
        - *Timer Resolution Latency* L^{timer}
- The timer resolution latency L^{timer} can often be much larger than the non-preemptable section latency L^{np}

The Timer Resolution Latency

- Kernel timers are generally implemented by using a hardware device that produces periodic interrupts
- Periodic timer interrupt → tick
- Example: periodic task (`setitimer()`, Posix timers, `clock_nanosleep()`, ...) τ_i with period T_i
- At the end of each job, τ_i sleeps for the next activation
- Activations are triggered by the periodic interrupt
  - Periodic tick interrupt, with period T^{tick}
  - Every T^{tick}, the kernel checks if the task must be woken up
  - If T_i is not multiple of T^{tick}, τ_i experiences a timer resolution latency

The Periodic Tick

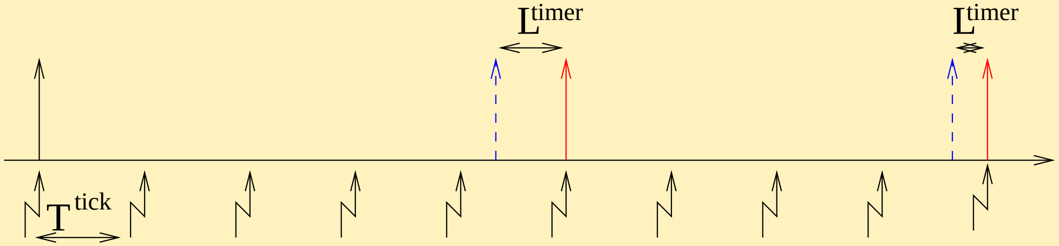
- Traditional operating systems: timer device programmed to generate a *periodic* interrupt
  - Example: in a PC, the Programmable Interval Timer (PIT) is programmed in *periodic mode*
- At every tick the execution enter kernel space
- The kernel executes and can
  - Wake up tasks
  - Adjust tasks priorities
  - Run the scheduler, when returning to user space → possible preemption
- The timer interrupt period is a trade-off between responsiveness (low latency) and throughput (low overhead)

Tick Tradeoff

- Large T^{tick} → large timer resolution latency
- Small T^{tick} → high number of interrupts
  - More switches between US and KS
  - Tasks are interrupted more often
  - $\Rightarrow$ Larger overhead
- For non real-time systems, it is possible to find a reasonable tradeoff
  - Linux 2.4: $10ms$ (HZ $=$ 100)
  - Linux 2.6: HZ $=$ 100, 250, or 1000
  - Other systems: $T^{tick} = 1/1024$

Timer Resolution Latency

- Experienced by all tasks that want to sleep for a specified time T



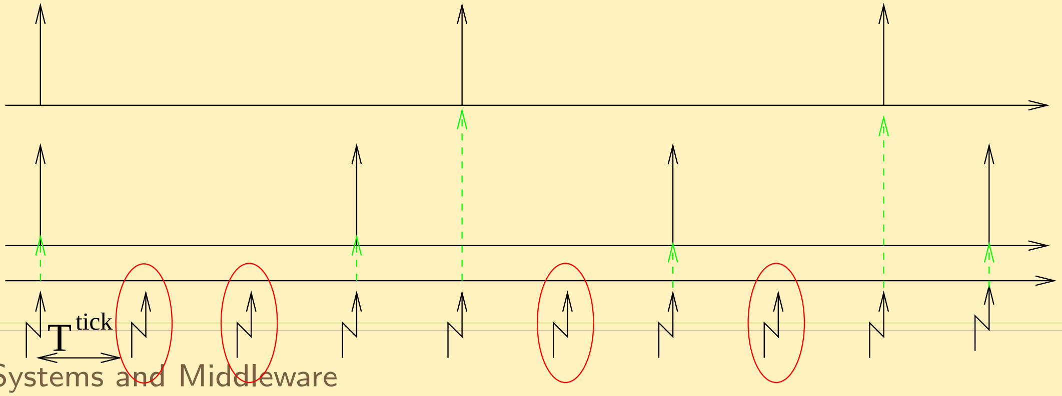
- τ_i must wake up at time $r_{i,j} = jT_i$
- But is woken up at time $t' = \left\lceil \frac{r_{i,j}}{T^{tick}} \right\rceil T^{tick}$
- So, the timer resolution latency is bounded:

$$L^{timer} = t' - r_{i,j} = \left\lceil \frac{r_{i,j}}{T^{tick}} \right\rceil T^{tick} - r_{i,j} =$$

$$= \left(\left\lceil \frac{r_{i,j}}{T^{tick}} \right\rceil - \frac{r_{i,j}}{T^{tick}} \right) T^{tick} \le T^{tick}$$

Problems with Periodic Ticks

- Reducing T^{tick} below $1ms$ is generally not acceptable...
- ...So, periodic tasks can expect a blocking time due to L^{timer} up to $1ms$
    - How large is the effect on the schedulability tests?
- Additional problems:
    - Tasks' periods are rounded to multiples of T^{tick}
    - Limit on the minimum task period: $\forall i, T_i \geq T^{tick}$
    - A lot of useless timer interrupts might be generated



Timers and Clocks

- Remember?
  - Timer: generate an event at a specified time t
  - Clock: keep track of the current system time
- A timer can be used to wake up a periodic task τ, a clock can be used to read the system time (`gettimeofday()`)
- **Timer Resolution**: minimum interval at which a periodic timer can fire
  - If periodic ticks are used, the timer resolution is T^{tick}
- **Clock Resolution**: minimum difference between two different times returned by the clock
  - What's the expected clock resolution?

Clock Resolution

- Traditional systems use a "tick counter" to keep track of the time
  - Very fast clock: return the number of ticks (jiffies in Linux) from the system boot
  - Clock Resolution: T^{tick}
- Modern PCs also provide higher resolution time sources...
  - For example, the TSC (TimeStamp Counter) on x86
  - High-Resolution clock: use the TSC (or higher resolution time source) for computing the time since the last timer tick...
- Summary: High-Resolution clocks **are easy**!
  - Every *modern* OS kernel provide them

Clock Resolution vs Timer Resolution

- Even using a “traditional” periodic timer tick, it is easy to provide high-resolution clocks
  - Time can be easily read with a high accuracy
- On the other hand, timer resolution is limited by the system tick T^{tick} (= `1 / HZ`)
  - It is impossible to generate events at arbitrary instants in time, without latencies

Timer Devices

- The timer device (example: the PIT - i8254 - on PCs) generally provides two operational modes: *periodic* and *one-shot*
- Programmed writing a value C in a counter register
- The counter register is decremented at a fixed rate
- When the counter is 0, an interrupt is generated
    - If the device is programmed in periodic mode, the counter register is automatically reset to the programmed value
    - If the device is programmed in one-shot mode, the kernel has to explicitly reprogram the device (setting the counter register to a new value)

Using the One-Shot Mode

- The periodic mode is easier to use! This is why most kernels use it
- When using one-shot mode, the timer interrupt handler must:
  1. Acknowledge the interrupt handler, as usual
  2. Check if a timer expired, and do its usual stuff...
  3. Compute when the next timer must fire
  4. Reprogram the timer device to generate an interrupt at the correct time
- Steps 3 and 4 are particularly critical and difficult

Reprogramming the Timer Device - 1

- When the kernel reprograms the timer device (step 4), it must know the current time...
- ...But the last known time is the time when the interrupt fired (before step 1)...
- Example:
  - A timer interrupt fires at time t_1
  - The interrupt handler starts (execution enters KS) at time t_1'
  - Before returning to US, the timer must be reprogrammed, at time t_1''
  - Next interrupt must fire at time t_2; the counter register is loaded with $t_2 - t_1$
  - Next interrupt will fire at $t_2 + (t_1'' - t_1)$

Reprogramming the Timer Device - 2

- The error described previously accumulates
- $\Rightarrow$ There is the risk to have a drift between real time and system time
- A *free run counter* which is not stopped at time t_1 is needed
- The counter is synchronised with the timer device $\Rightarrow$ the value of the counter at time t_1 is known
- This permits to know the time t_1'' $\Rightarrow$ the new counter register value can be computed correctly
- On a PC, the second PIT counter, or the TSC, or the APIC timer can be used as a free run counter
- Final note: reprogramming the PIC is an expensive operation $\Rightarrow$ it is better to use other timer devices

High Resolution Timers

- Serious real-time kernels implement *High-Resolution Timers* programming the device in one-shot mode
    - Already implemented in RT-Mach
    - Also implemented in RTLinux, Resource Kernels, RTAI, SHaRK, etc…
- General-Purpose kernels are more concerned about stability and overhead
- Some techniques have been proposed to reduce the overhead
    - Soft Timers
    - Firm Timers
- HRT entered the Linux kernel in version 2.6.21

HRT and Timer Ticks

- Compatibility with “traditional” kernels:
  - The tick event can be emulated through high-resolution timers
  - ⇒ Timer device programmed to generate interrupts both:
    - When needed to serve a timer, and
    - At tick boundaries
- ...But the “tick” concept is now useless
  - Tickless (or `NO_HZ`) system
  - Good for saving power
    - In some lucky situations, average of 1 timer interrupt per second!
  - The implementation still has some limitations, but is possible in theory

Some Notes on Linux Timers

- Terminology:
  - Timer → Clock Event Source
- Traditional architecture:
  - Clocks and clock event sources are “scorrelated”
  - Implemented in architecture code (`linux/arch/xxx/kernel/...`) ⇒ lot of code duplication
- The (architecture dependend) clock event source code provides periodic ticks invoking generic (`linux/kernel`) code that:
  - Performs process execution time accounting
  - Increase the system *jiffies*
  - Handles system timers

Linux Timers Handling

- System timers stored in a *timer wheel* structure...
    - Optimized for insertion / extraction (O(1))
    - Scales well with the number of timers
- Periodic check for expired timers can be inefficient
    - Structure based on a set of arrays
    - The first timers to expire are in the *base array*
    - When a time expire it might be necessary to move timers from an array to the previous one (*timers cascading*)
- See `linux/kernel/timer.c`
- Cascading works well when a lot of timers expire together (timers clustering - on a tick boundary)

Efficient High-Resolution Timers

- Timer wheel → inefficient in storing / handling high-resolution timers
  - High resolution timers tend to expire “too often” (no clustering)
- Some form of clustering is needed for supporting efficient structures
  - Dedicated real-time systems do not care, but Linux **must** have a scalable timers subsystem
  - Early high-resolution timers implementations on Linux (KURT, Montavista high-res timers, etc...) failed on this
- A distinction between timers that need high resolution and timers that can be clustered helps...

Timers and Timeouts

- Most of the system timers really are timeouts
  - Used to detect anomalies and error conditions
  - Do not fire in general
  - Must be possible to efficiently insert and remove them from the timer list
  - Do not need high resolution (can be clustered)
- Other timers need high resolution
  - They generally expire
  - No need to efficiently remove them from the timer list

HRTimers in Linux

- hrtimers: Rework the timer wheel to allow efficient handling of high-resolution timers
- GTOD (Generic Time of Day): rework the *clock* subsystem moving most of the code from architecture-dependent to generic code
  - Remove code duplication
  - Remove dependency on periodic tick
- clockevents: generic (non arch-dependent) infrastructure for handling clock event sources
  - Remove code duplication
  - Make it possible to reprogram the timer device