The Timer Resolution
Latency

Real Time Operating Systems and Middleware

Luca Abeni
luca.abeni@unitn.it
Latency

- Latency: measure of the difference between the theoretical and actual schedule
  - Task $\tau$ expects to be scheduled at time $t$ . . .
  - . . . but is actually scheduled at time $t'$
  - $\Rightarrow$ Latency $L = t' - t$

- The latency $L$ can be modelled as a blocking time
  - $\Rightarrow$ affects the guarantee test
    - Similar to what done for shared resources
    - Blocking time due to latency, not to priority inversion
Effects of the Latency

- Upper bound for $L$? If not known, no schedulability tests!!!
  - The latency must be bounded: $\exists L_{\text{max}} : L < L_{\text{max}}$
- If $L_{\text{max}}$ is too high, only few task sets result to be schedulable
  - Large blocking time experienced by all tasks!
  - The worst-case latency $L_{\text{max}}$ cannot be too high
Sources of Latency

- A task \( \tau_i \) 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:
  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 \( \rightarrow \) let’s not consider it

- \( L^2 \) is the **non-preemptable section latency** (\( L^{np} \))

- \( L^1 \) is due to the delayed interrupt generation
Non-Preemptable Section Latency

- Delay between time when an event is generated and when the kernel handles it
  - Due to non-preemptable sections in the kernel, which delay the response to hardware interrupts
  - Composed by various parts: interrupt disabling, bottom halves delaying, . . .

- Depends on how the kernel handles the various events...

- Will talk about it later!
Interrupt Generation Latency

- Hardware interrupts: generated by devices
- Sometimes, an interrupt should be generated at time $t$ . . .
- . . . but it is actually generated 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

- Interrupt generation latency for a hw timer device

\[ L_{\text{timer}} \] can often be much larger than the non-preemptable section latency \[ L_{np} \]

- Where does it come from?
  - Kernel timers are generally implemented by using a hardware device that produces periodic interrupts

- Can we do anything about it?
• Periodic timer interrupt $\rightarrow$ tick

• Example: periodic task (\texttt{setitimer()}, Posix timers, \texttt{clock_nanosleep()}, \ldots) $\tau_i$ with period $T_i$

• Job end $\rightarrow \tau_i$ sleeps for the next activation

• Activations are triggered by the periodic interrupt
  
  • Periodic tick interrupt, with period $T^{\text{tick}}$
  
  • Every $T^{\text{tick}}$, the kernel checks if the task must be woken up
  
  • If $T_i$ is not multiple of $T^{\text{tick}}$, $\tau_i$ 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 enters kernel space
- The kernel executes and can
  - Wake up tasks
  - Adjust tasks priorities
  - Run the scheduler, when returning to user space → possible preemption
Tick Tradeoff

- Timer interrupt period: trade-off between responsiveness (low latency) and throughput (low overhead)

- Large $T^{\text{tick}}$ → large timer resolution latency

- Small $T^{\text{tick}}$ → high number of interrupts
  - More switches between US and KS
  - Tasks are interrupted more often
  - $\Rightarrow$ Larger overhead
Trade-off Examples

- For non real-time systems, it is possible to find a reasonable tradeoff...

- But it still depends on the workload!
  - Desktop or server?

- Example: the Linux kernel
  - 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 $T$

- $\tau_i$ must wake up at time $r_{i,j} = jT_i$

- But is woken up at time $t' = \left\lceil \frac{r_{i,j}}{T_{\text{tick}}} \right\rceil T_{\text{tick}}$
The timer resolution latency is bounded:

\[ t = r_{i,j} \]

\[ t' = \left\lceil \frac{r_{i,j}}{T_{\text{tick}}} \right\rceil T_{\text{tick}} \]

\[ L_{\text{timer}} = t' - r_{i,j} = \left\lceil \frac{r_{i,j}}{T_{\text{tick}}} \right\rceil T_{\text{tick}} - r_{i,j} = \]

\[ = \left( \left\lceil \frac{r_{i,j}}{T_{\text{tick}}} \right\rceil - \frac{r_{i,j}}{T_{\text{tick}}} \right) T_{\text{tick}} \leq T_{\text{tick}} \]
Problems with Periodic Ticks

- Reducing $T^{\text{tick}}$ below $1\text{ms}$ is generally not acceptable.

- So, periodic tasks can expect a blocking time due to $L^{\text{timer}}$ up to $1\text{ms}$

- How large is the effect on the schedulability tests?

- Additional problems:
  - Tasks’ periods are rounded to multiples of $T^{\text{tick}}$
  - Limit on the minimum task period: $\forall i, T_i \geq T^{\text{tick}}$

- ...
• Additional problem: 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 $\tau$, a clock can be used to read the system time ($\text{gettimeofday}()$)

- Timer Resolution

- Clock Resolution
Timer and Clock Resolution

- **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 OSs use a “tick counter”
  - Very fast clock: return the number of ticks (jiffies in Linux) from the system boot
  - Clock Resolution: $T_{\text{tick}}$
- Modern PCs have higher resolution time sources...
  - On x86, TSC (TimeStamp Counter)
  - High-Resolution clock: use the TSC to compute 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 $T^{\text{tick}} (= 1 / \text{HZ})$
  - It is impossible to generate events at arbitrary instants in time, without latencies
Timer Devices

- Timer devices (ex: PIT - i8254) generally work in 2 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.
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)...

- A timer interrupt fires at time $t_1$
- The interrupt handler starts (enter 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 $t_2 + (t''_1 - t_1)$
The error described previously accumulates

\[ \Rightarrow \text{Risk: drift between real time and system time} \]

A \textit{free run counter} (not stopped at \( t_1 \)) 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
High Resolution Timers

- Serious real-time kernels \(\rightarrow\) **High-Resolution Timers**
  (use hw timer in one-shot mode)
  - Already implemented in RT-Mach
  - Also implemented in RTLinux, RTAI and others

- General-Purpose kernels are more concerned about stability and overhead
  - Too much overhead for GP kernels?

- Fixed: hrtimers are in Linux since version 2.6.21
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