Integrated Schedulers for a Predictable Interrupt Management on Real-Time Kernels

S. Sáez   A. Crespo

Instituto de Automática e Informática Industrial
Universidad Politécnica de Valencia

19th International Conference on Reliable Software Technologies
June 2014, Paris

Index

Index

Event-driven real-time schedulers

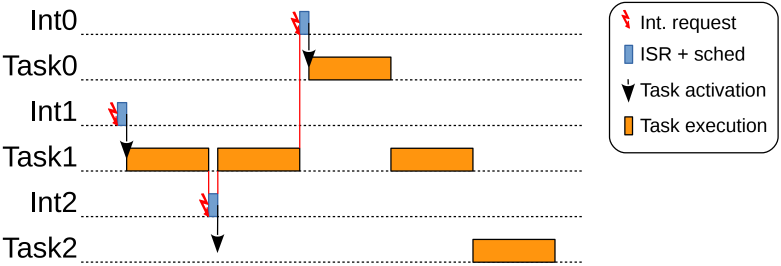
- Activate tasks when certain events occur:
  - external interrupts, timer interrupts, software mechanisms, ...
- Depending on the activation event, tasks can be classified in:

  Hardware Activated Tasks
  : Tasks are woken up by *Interrupt Service Routines*.

  Software Activated Tasks
  : Tasks are woken up by software mechanisms: timing events, delays, semaphores, barriers, ...
- Tasks have priorities that establish the execution order.
- These priorities can be used to map task's criticality level.
- It is desirable that a high priority task did not suffer unnecessary interference from lower priority tasks.

Classical Interrupt Model

- Since interrupts have higher priorities than any task, their ISRs are always executed despite of:
  - the priority of the task currently in execution
  - the task that is going to be activated by a given interrupt.



- High priority tasks suffer interference from interrupts used to activate lower priority tasks that are obviously not executed at activation instant.

Classical Interrupt Model (cont.)

- Blocking Time at priority i is:

$$B(i) = \underbrace{|L(i)|}_{\text{Nr of lower priority tasks}} \times \underbrace{(\delta^{\text{isr}} + \delta^{\text{Sched}-\text{A}})}_{\text{Task activation overhead}}$$

- $\delta^{\text{isr}} \rightarrow$ mostly ISR execution time
- $\delta^{\text{Sched}-\text{A}} \rightarrow$ scheduler has to ...

  $Q^{R}_{\text{insert}} \rightarrow$ update the *ready queue*,

  $Q^{R}_{\text{find}-\text{min}} \rightarrow$ determine the highest priority task

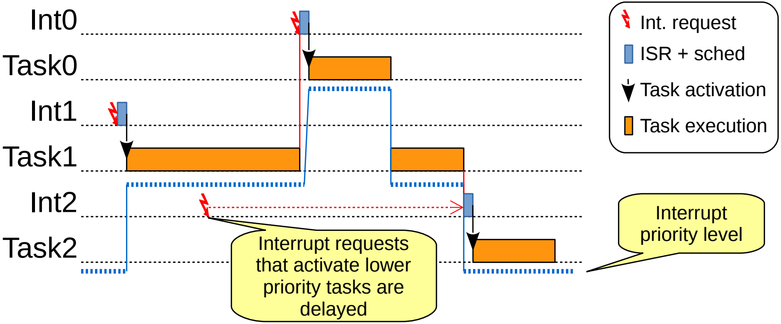
  ... each time a task is activated.

Question:
If the activated task is not going to be immediately executed, why does the ISR have to interfere with high priority tasks?

Index

Integrated Interrupt Model

- A unique priority space is used for tasks and ISRs.
- Each time a task is activated, the interrupt priority level is changed to avoid lower priority interrupts to be attended.

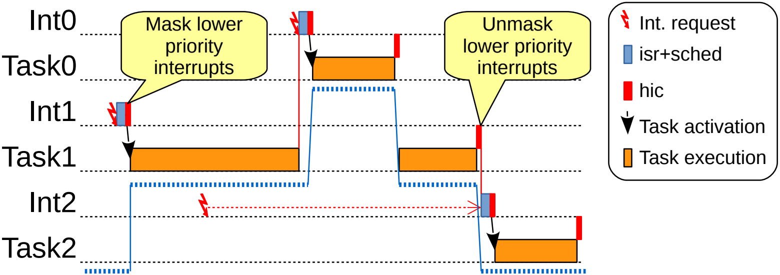


- The *Hardware Interrupt Controller* receives lower priority interrupt requests but the CPU is not notified.

Integrated Interrupts Model: Hardware Activated Tasks

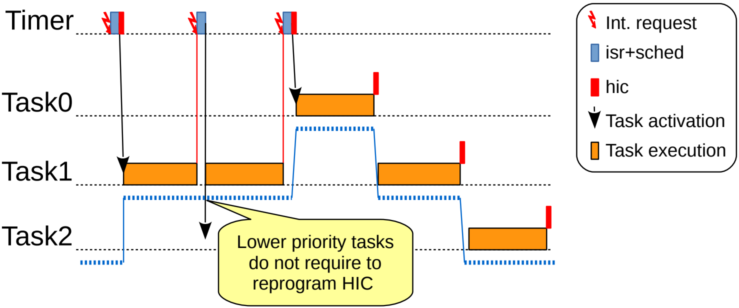
- Requires a Hardware Interrupt Controller with multiple interrupt priorities.
- If this HW support is not present, an additional overhead is introduced:

  δ^{hic} time to mask unnecessary interrupts according to the new priority level.



Integrated Interrupts Model: Software Activated Tasks

- To fully support SAT multiple HW timers with different priorities are required.
  - ⇒ This HW is not commonly available.
- SATs have to share the Timer interrupt
  - ⇒ Timer interrupt is always enabled.



Integrated Interrupts Model: Software Activated Tasks

- Only SATs activations can produce interference, but the interference is higher.

→ ∃ Blocking Time at priority i due to unnecessary activations:

$$B^{\mathrm{SAT}}(i) = |L_{\mathrm{SAT}}(i)| \times (\delta^{\mathrm{isr}} + \delta^{\mathrm{Sched-A}} + \delta^{\mathrm{timer}})$$

$$\delta^{\mathrm{Sched-A}} = \overbrace{Q^{W}_{\mathrm{delete-min}}}^{\text{Remove } \tau \text{ from timer queue}} + \overbrace{Q^{W}_{\mathrm{find-min}}}^{\text{Find out next activation time}} + \underbrace{Q^{R}_{\mathrm{insert}}}_{\text{Insert } \tau \text{ into ready queue}} + \underbrace{Q^{R}_{\mathrm{find-min}}}_{\text{Schedule}}$$

δ^{timer} → Overhead of reprogramming the HW timer.

Index

A new proposal: Virtual Integrated Interrupt Model

GOAL: To program timer interrupts corresponding **only** to higher priority tasks.

- Each time a task starts its execution, it has to:
  1. Find the closer waiting task with a priority higher than the current one.
     $\Rightarrow$ **The next preemptor**.
  2. Program the timer interrupt for activating only the next preemptor.
- When the next preemptor wakes up, previously ignored lower priority tasks are also awakened.

Drawbacks

→ This approach gives rise to additional scheduling overheads.

→ Commonly used data structures for ready and waiting queues are not adequate for these new scheduling operations.

Scheduling overheads using ready/waiting queues

- Waiting queue is sorted by release time
  $\rightarrow$ to find the next preemptor could have a $O(N^{\mathrm{W}})$ cost.
- When the next preemptor is activated
  $\rightarrow$ N_p lower priority tasks have to be moved from the waiting queue into the ready queue.
  $\rightarrow$ in the worst case $N_p = N^{\mathrm{W}}$.

$$\begin{aligned}\delta_I^{\mathrm{Sched-A}} &= N_p \times \overbrace{(Q^W_{\mathrm{delete-min}} + Q^R_{\mathrm{insert}})}^{\text{wake up a lower priority task}} \\ &+ \underbrace{Q^R_{\mathrm{find-min}}}_{\text{select the next task}} + \underbrace{Q^W_{\mathrm{find-preemptor}}}_{\text{determine its next preemptor}}\end{aligned}$$

Index

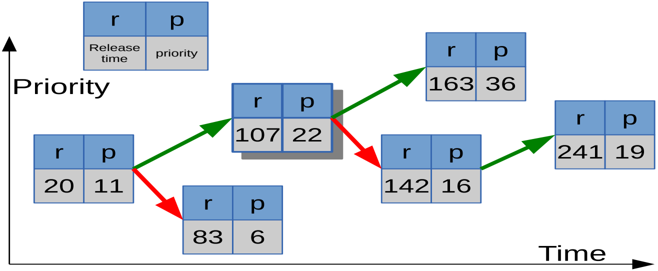
Scheduling Cartesian Tree

Goals

- To avoid massive task movements from waiting queue.
- To efficiently determine the next preemptor.

Cartesian Tree

- A binary tree sorted by two keys:
  priority (top-bottom subtrees) and release time (node depth).



Scheduling Cartesian Tree (cont.)

- Only one tree that represents the ready and waiting queue.
- SC-Tree is sorted by absolute release time:
  $\Rightarrow$ a task activation does not modify the structure.

$$\delta_I^{\mathrm{Sched-A}} = \cancel{N_p \times (Q^W_{\mathrm{delete-min}} + Q^R_{\mathrm{insert}})} + Q^R_{\mathrm{find-min}} + Q^W_{\mathrm{find-preemptor}}$$

- Next preemptor is the *top* child node:
  $\Rightarrow$ activation time overhead is constant. **No release jitter!!**

$$\delta_I^{\mathrm{Sched-A}} = C_{\mathrm{find-min}} + C_{\mathrm{find-preemptor}}$$

- Main scheduling overhead occurs during task suspension.
  - It can be accounted as part of the WCET.
  - A careful implementation could allow preemptive SC-Tree operations
    $\rightarrow$ It produces no/low blocking times.

Priority Inheritance

- When a task locks/unlocks a shared resource and an Inheritance Protocol is used, priority changes are produced.
- These temporal priority changes have an aditional cost in an Integrated Interrupt Model:

$$
\begin{aligned}
Q^{PO} &= \delta^{\text{hic}} + Q^{R}_{\text{insert}} + Q^{R}_{\text{delete-min}} \\
&+ \cancelto{C_f}{2 \times Q^{R}_{\text{find-min}}} + \cancelto{C_p}{2 \times Q^{W}_{\text{find-preemptor}}}
\end{aligned}
$$

- When the shared resource is freed, the cost of activate pending medium priority tasks is lower than if they had been activated in their release instants:
  - $\rightarrow$ **no** ISR has been executed
  - $\rightarrow$ **no** interrupt priority level has been changed.

Index

Summarising

- A new approach has been presented to completely avoid activation interference from lower priority tasks.
- Suitability of dual-queue schedulers to implement this approach has been evaluated.
- A new data structure has been proposed and its overhead compared against the classical model.
- The paper provides the necessary tools to check if this approach is suitable for a given system taking into account the real system overheads.

Pending issues

- SC-Tree behaviour during priority inheritance can be improved.
- To study the applicability of the Integrated Interrupt Model to dynamic priorities.

Integrated Schedulers for a Predictable Interrupt Management on Real-Time Kernels

S. Sáez A. Crespo

Instituto de Automática e Informática Industrial
Universidad Politécnica de Valencia

19th International Conference on Reliable Software Technologies
June 2014, Paris

Thank you! Any question?