Real-Time Systems

Anno accademico 2009/10
Laurea magistrale in informatica
Dipartimento di Matematica Pura e Applicata
Università di Padova
Tullio Vardanega

Outline

1. **Introduction**
2. **Dependability issues**
3. **Scheduling issues**
4. **More on fixed-priority scheduling**
5. **Task interactions and blocking**
6. **System issues**
7. **Multi-cores and distribution**

Bibliography

- J. Liu, "Real-Time Systems", Prentice Hall, 2000
- A. Burns, A. Wellings, “Concurrent and Real-Time Programming in Ada”, Cambridge University Press, 2007
- A. Burns, A. Wellings, "Real Time Systems and Programming Languages: Ada 95, Real-Time Java and Real-Time C/POSIX", Addison-Wesley, 2009

6.a System issues

Context switch

- Preemption causes time and space overheads which should be duly accounted for in schedulability tests
- Under preemption every single job incurs at least two context switches
  - One at activation
  - One at completion
- The resulting costs should be charged to the job
  - Knowing the timing behavior of the run-time system we could incorporate overhead costs in schedulability tests

Priority levels – 1

- The FPS techniques that we have studied assume jobs to have *distinct* priorities
- It is not obvious however that we can always meet this requirement
  - Jobs may have to share priority levels
  - At the same level of priority, selection may be FIFO or round-robin
- If priority levels are shared we have a worst-case situation to contemplate in the analysis
  - That job J_i be released immediately *after* all other jobs residing at its level of priority

Priority levels – 2

- Let $T_\varepsilon(i)$ denote the set of jobs with priority equal to J_i excluding J_i itself
- The time demand equation for J_i to study in the interval $0 < t \leq \min(D_i, p_i)$ then becomes

$$W_{i,1}(t) = e_i + b_i + \Sigma_{j \in T_\varepsilon(i)} e_j + \Sigma_{(k=1,\ldots,i-1)} \lceil t/p_k \rceil e_k$$

- This obviously worsens J_i's response time
  - But at system level the impact in terms of schedulability loss may not be as bad

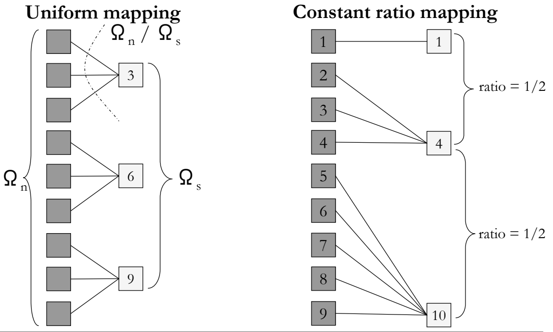
Priority levels – 3

- When the number $[1,..,\Omega_n]$ of *assigned priorities* is greater than the number $[\Pi_1,.., \Pi_{\Omega s}]$ of *available priorities* (priority grid) then we need some Ω_n–to–Ω_s mapping
  - All (top-range) assigned priorities $\geq \Pi_1$ take value Π_1
  - Those in the interval $(\Pi_{k-1}, \Pi_k]$ take value Π_k progressing in the interval $1 < k \leq \Omega_s$
- Two main techniques
  - **Uniform mapping**
  - **Constant ratio mapping** [Lehoczky & Sha, 1986]

Priority levels – 4

- **Uniform mapping**
  - Availability is uniformly apportioned to needs
  - $Q = \lfloor \Omega_n / \Omega_s \rfloor \Rightarrow \Pi_k = kQ$ for $k=1,2,\ldots,\Omega_s-1$ and $\Pi_{\Omega s} = \Omega_n$
  - **Example**: from $\Omega_n=9$ and $\Omega_s=3$ we obtain $\Pi_1=3$, $\Pi_2=6$, $\Pi_3=10$, whence 1-3 $\rightarrow \Pi_1$, 4-6 $\rightarrow \Pi_2$, 7-9 $\rightarrow \Pi_3$
- **Constant ratio mapping**
  - Keeps the ratio $(\Pi_{i-1}+1)/\Pi_i$ constant for $i=2,\ldots,\Omega_s$ for the better good of higher-priority jobs
  - **Example** (same case as above): $\Pi_1=1$, $\Pi_2=4$, $\Pi_3=10$ with ratio at ½, whence 1 $\rightarrow \Pi_1$, 2-4 $\rightarrow \Pi_2$, 5-9 $\rightarrow \Pi_3$

Priority levels – 5



Priority levels – 6

- Lehoczky & Sha showed that the use of constant ratio mapping degrades the schedulable utilization of the RM scheduling algorithm gracefully
  - For large *n* with $D_i = p_i$ for all *i*, and g denoting the minimum ratio in the given priority grid
  - Schedulable utilization f(g) evalues to
    - ln(2g)+1-g for g>1/2
    - g for g≤1/2
  - The f(g)/ln 2 ratio is termed ***relative schedulability***
  - **Example**: with $\Omega_s = 256$ and $\Omega_n = 100.000$, relative schedulability evaluates to 0,9986
- It follows that 256 priority levels suffice for RM scheduling

Tick scheduling – 1

- So far we have tacitly assumed that the scheduler operates on an *event-driven* basis
  - The scheduler always immediately executes upon the occurrence of a ***scheduling event***
  - If it was so then we could reasonably assume that a job is placed in the ready queue at its release time
- In actual fact the scheduler may also operate in a *time-driven* fashion
  - In that case the scheduling decisions are made and executed periodically on the arrival of ***clock interrupts***
  - This mode of operation is termed ***tick scheduling***

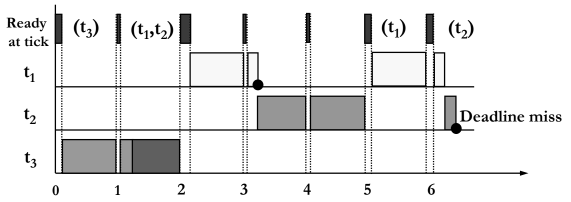
Tick scheduling – 2

- With tick scheduling the time at which the scheduler acknowledges a job's release time may be delayed by 1 clock interrupt
  - This delay may have negative impact on the job's response time
  - Hence we must assume a logical place where jobs in the "release time arrived but not yet acknowledged" state are held
    - The time and space overhead of transferring jobs from that logical place to the ready queue is not null and must be accounted for in the schedulability test
    - Together with the time and space overhead of handling clock interrupts

Example

T = {t_1=(0.1, 4, 1, 4), t_2=(0.1, 5, 1.8, 5), t_3=(0, 20, 5, 20)}
t_3's first section not preemptable and with duration 1.1

From RTA with event-driven scheduling we have R_1= 2.1, R_2= 3.9, R_3= 14.4 (OK)
What with tick scheduling, clock period 1 and time overhead 0.05 + (0.06 * n) ?



Tick scheduling – 3

- The effect of tick scheduling is captured in the RTA for job J_i
  - By introducing a notional task $T_0 = (p_0, e_0)$ at the highest priority to account for the cost of handling clock interrupts
  - For all jobs J_k at priority greater than or equal to J_i, by adding to e_k the time overhead m_0 due to moving them to the ready queue
    - $(K_k + 1)$ times for the K_k times that job J_k may self suspend
  - For all jobs J_l at priority lower than J_i, by introducing a notional task (p_l, m_0), for every such job to account for the time overhead of moving them to the ready queue
  - Computing $b_i(np)$ as a function of p_0 as J_i may suffer up to p_0 units of delay after becoming ready already without non-preemptable execution and thus $b_i(np) = (\lceil \max_k (\theta_k / p_0) \rceil + 1)\, p_0$ including non-preemption
    - Where θ_k is the maximum time of non-preemptable execution by any job J_k

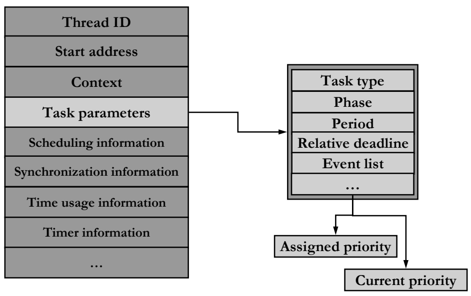
Real-time operating systems – 1

- Must be small, modular, extensible
  - **Small footprint** because there are often stored in ROM (which used to be little) and because most embedded systems have little RAM
    - Real-time embedded systems do not include permanent storage other than for background aperiodic activities
  - **Modular** because this facilitates verification, validation and certification of its design and implementation, including of temporal predictability
  - **Extensible** because some but not all specific systems may need functionalities above and beyond the core ones
- Adhering to the principle of microkernel architecture
  - Minimal kernel services include scheduling, inter-process communication and synchronization, interrupt handling

Real-time operating systems – 2

- Tasks must be known to the RTOS
  - Tasks (processes, threads) are the unit of CPU allocation by the scheduler
    - Tasks issue jobs, one at a time, which are subject to scheduling and dispatching
  - Upon creation of a task, some memory is assigned from RAM to create the ***Task Control Block*** for that task
  - The insertion of a task in a state queue (e.g., ready queue) is made by placing a pointer to the relevant TCB
  - The disposal of a task at end of life requires removal of its TCB and de-allocation of any memory it had in use
    - In typical embedded systems, tasks *never terminate*

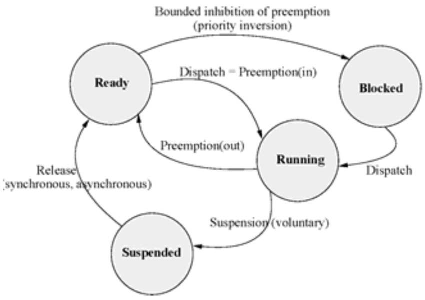
Task control block



Real-time operating systems – 3

- For better generality tasks are often realized at *application level* instead of as primitive entities of the RTOS
- **Periodic task**
  - An RTOS thread that hangs on a suspension point which is periodically released; after release it executes application-specific code (corresponding to issuing a job) and then returns to the suspension point
- **Sporadic task**
  - An RTOS thread whose suspension point is not released periodically but with bounded minimum distance and that after release issues its job ands then returns to the suspension point
- **Aperiodic task**
  - Indistinguishable from the other tasks other than for the absence of deadline (because of which it executes in the background)

Task states – 1



Task states – 2

- Tasks enter the ***suspended*** state only voluntarily
  - By making a primitive invocation that causes them to hang on a periodic / sporadic suspension point
- The RTOS needs specialized structures to handle the distinct forms of suspension
  - A time-based queue for periodic suspensions
  - An event-based queue for sporadic suspensions
    - But someone shall still take care of warranting minimum separation between subsequent releases (**!**)

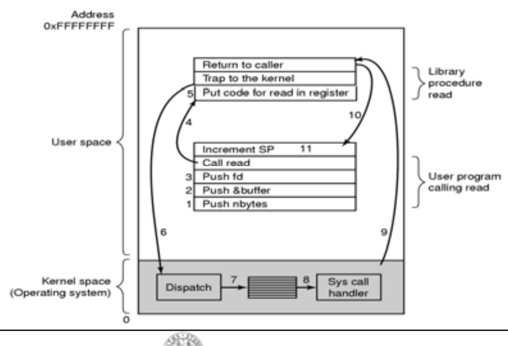
System calls – 1

- The most part of RTOS services are executed in response to direct or indirect invocations by tasks
  - These invocations are termed **system calls**
- System calls are not directly visible to the application
- They are hidden in procedure calls exported by compiler libraries
  - The library call does all of the preparatory work needed to make the correct invocation of the actual system call on behalf of the application

System calls – 2

- In embedded systems the RTOS and the application share memory
  - Not the case in general-purpose operating systems
  - Real-time embedded applications can be better trusted and we do not want to pay the space and time overhead that arises from address space separation
  - This however implies that the RTOS must protect its own data structures from the risk of race condition

System calls – 3



The scheduler – 1

- This is a distinct part of the RTOS that does not execute in response to application invocation
- It acts every time a task changes state
  - The corresponding time events are termed ***dispatching points***
- Scheduler "activation" is often periodic in response to ***clock interrupts***
  - Not only with tick scheduling

The scheduler – 2

- At every clock interrupt the scheduler must
  - Manage the queue of time-based events pending
  - Increment the execution time budget counter of the running job to support the time-based scheduling policy in force (e.g., round-robin)
  - Manage the ready queue
- The 10 ms or above period (***tick size***) typical of general-purpose operating systems is not fit for RTOS
  - But a higher frequency may incur excessive overhead
- The scheduler needs to make provisions for event-driven execution too

I/O issues

- The I/O subsystem of a real-time system may require its own scheduler
- Simple methods to access an I/O resource
  - Use a non-preemptive FIFO policy
  - Use some kind of TDMA scheme
- Preemptive scheduling techniques as those in use for processor scheduling
  - For instance, RM, EDF, LLF

  can be used to schedule I/O requests

Interrupt handling – 1

- Hardware interrupts are the most efficient manner for the processor to notify the application about the occurrence of external events
  - E.g., the completion of asynchronous I/O operations
- Frequency and computational load of the interrupt handling activities vary with the interrupt source
- For reasons of efficiency the interrupt handling service is typically subdivided in an ***immediate*** part and a ***deferred*** part
  - The immediate part executes at the level of interrupt priority, above all software priorities
  - The deferred part executes as a normal software activity

Interrupt handling – 2

- When the hardware interface asserts an interrupt the processor saves the PC and PSW registers in the interrupt stack and jumps to the address of the relevant interrupt service routine (ISR)
  - At this time interrupts are disabled to prevent race conditions from happening on the arrival of further interrupts
  - Interrupts arriving at that time may be lost or just kept pending depending on the hardware capability
  - Interrupts operate at an *assigned level of priority*
  - The interrupt source may be determined by ***polling*** or via an ***interrupt vector***
    - Polling is hardware independent hence more generally applicable but it increases ***latency*** of interrupt service
    - Vectoring needs specialized hardware but it contains latency
- As these actions complete, registers are restored and interrupts are enabled again

Interrupt handling – 3

- The worst-case *latency* incurred on interrupt handling is determined by the time needed to
  - Bring the current instruction to completion, save registers, clear the pipeline, acquire the interrupt vector, activate the trap mechanism
  - Disable interrupts
  - Complete the (remaining) execution of the ISR at higher priority
    - This duration corresponds to interference across interrupts
  - Save the context of the interrupted task, identify the interrupt source and jump to the corresponding ISR
  - Begin execution of the selected ISR
    - Interrupt service can have a ***device-independent*** part and a ***device-specific*** part

Interrupt handling – 4

- To reduce distributed overhead, the deferred part of the interrupt handling service must be preemptable
  - Hence it must execute at software priority
- But it still may directly or indirectly operate on RTOS level data structures
  - Those structures must be therefore protected by appropriate access control protocols
  - If we can do that then we do not need the RTOS to spawn its own tasks for this purpose

Interrupt handling – 5

- To achieve better responsiveness for interrupt services schemes such as ***slack stealing*** or ***bandwidth preservation*** could be used
  - Bandwidth preservation keeps the reserve of execution budget not used by aperiodic activities in between periodic replenishments
  - But their implementation needs specialized support from the RTOS

Time management – 1

- A system clock consists of
  - A periodic counting register
    - Automatically reset to the ***tick size*** every time it reaches the ***triggering edge*** and triggers the ***clock tick***
  - The register a *hardware part* automatically decremented at very clock pulse and a *software part* incremented by the handler of the clock tick
  - A queue of time events fired in the interval, whose treatment is pending
  - An (immediate) interrupt handling service

Time management – 2

- The frequency of the clock tick determines the ***resolution*** (granularity) of the software part of the clock
  - The resolution should be an integer divisor of the tick size
  - So that the RTOS may perform tick scheduling at every N clock ticks
  - Then we have more frequent ***time-service interrupts*** and less frequent (1/N) ***clock interrupts***

Time management – 3

- The resolution of the software clock is an important design parameter of an RTOS
  - The finer the resolution the better the clock accuracy but the larger the time-service interrupt overhead
- There is a delicate balance between the clock accuracy needed by the application and the clock resolution that can be afforded by the system
  - There is intrinsic latency in any query made by a software task to the software clock
    - E.g., 439 clock cycles in ORK for the Leon microprocessor, corresponding to about 11 microseconds at 40 MHz
- The resolution cannot be finer-grained than the maximum latency that may be incurred in accessing the clock

Time management – 4

- Beside periodic clocks RTOS must support ***one-shot timers*** a.k.a. *interval timers*
  - They operate in a programmed (non-repetitive) way
- The RTOS scans the queue of the programmed time events to set the time of the next interrupt due from the interval timer
  - The resolution of the interval timer is limited by the time overhead of its handling by the RTOS
    - E.g., 7.061 clock cycles in ORK for Leon

Time management – 5

- The accuracy of time events is given by the difference between the time at which the event occurred and the time value as programmed
- Depends on three fundamental factors of influence
  - The frequency at which the time-event queues are inspected
    - If interval timers were not used, this would correspond to the period of time-service interrupts
  - The policy with which the RTOS handles the time-event queues
    - LIFO vs. FIFO
  - The time overhead cost of handling time events in the queue
- The release time of periodic tasks is inherently exposed to jitter

Summary

- RTOS design issues
- Context switch
- Priority levels
- Tick scheduling
- Interrupt handling
- Time management