Off-line Scheduling; Time Triggered and Event Triggered;

Gerhard Fohler

Mälardalen University, Sweden

gerhard.fohler@mdh.se

Activation Paradigms

- activation of activities - tasks
  - when are events recognized?
  - who initiated activities?
  - when are decisions taken?
- event triggered – ET
  - event initiates activities in system immediately
- time triggered – TT
  - activities initiated at predefined points in time

Time triggered - Rationale

activities initiated at predefined points in time

everything planned before system is deployed

How?

offline scheduling - scheduling table

  - complex algorithm
  - retries possible

- slots – time triggered activation of dispatcher
  - period of dispatcher minimum granularity in system
- runtime dispatcher executes decision in table

Which cost?

“everything planned before system is deployed”

- need to know everything
  - all environmental situations
    …and time of occurrence
  - all task parameters
    …including arrival times
  - all system parameters
    …for entire lifetime
- very high cost

Which benefit for that price?

"everything planned before system is deployed"

- *know everything* before runtime
  - schedulability test
    - implicit in offline schedule, "constructive proof"
    - not proven that there is *no* situation where timing could be violated, but show that in *this one* are met
    - complex demands, distributed, end-to-end, jitter, …
  - testing, certification
    - test space reduced dramatically
    - *deterministic,* i.e., know exact what is going on when
- low runtime overhead
  - very simple runtime dispatching, table (list) lookup

- simple fault-tolerance
  replica determinism
- network
  receiver based error detection
- non temporal constraints easy to integrate
  - energy, cost
- high resource utilization
  - no pessimism in scheduling overhead
    don’t have to assume worst case, but know actual case

Issues

“everything planned before system is deployed”

- anything that is not *completely known* cannot be handled at all
- zero flexibility
- assumes periodic world
- pessimism due to worst case assumptions, lack or reclaiming

MARS, TTP, TU Vienna, TTTECH Kopetz et. al.

How long to schedule?

- standard OS schedulers work on strategies without guarantees
  - handle “task transition graph” waiting - ready - executing…
  - select one out of the ready tasks to execute
  - perhaps prevent deadlocks etc.
  - go on until shutdown or system lock/crash, e.g., windows
- off-line guarantees: before, for entire *mission lifetime*
  - minutes
  - hours, days, more
  - need to guarantee every one of them
  - combinatorial explosion

Shorten analyzed lifetime

- analyze only single, selected part of lifetime
  - worst case proofs
  - need to ensure assume worst case is worst case
    - restrict complete freedom of task parameters
    - periods
- analyze repeating patterns during lifetime
  - typically periods
  - if harmonic, enough to analyze for duration of longest period
  - if not, *least common multiple LCM* of all involved periods
  - can be large
  - execute repeatedly

How to schedule within LCM?

- *Cyclic scheduling*
  - tasks in period classes
  - schedule tasks within classes
  - group task class schedules
  - …until all tasks scheduled

- easy to handle, historically popular

very different from offline scheduling!

less powerful, more restrictive, etc

often mixed up

- *off-line scheduling*
  static, pre run-time
  - construct schedule of length LCM
  - apply smart method
  - fulfill all constraints
  - not limited to “period concatenation”

Off-line Scheduling Methods

What do we want to achieve?

- we want to find solutions
  - NP hard in more than trivial cases
    can take very long time
- have to optimize search to find solutions fast

but

- once we find solution, we are done
- likely that first try will not work, maybe solution does not exist
- what if we don't find one/does not exist?
- total time spent in schedule design:
  time of not (finding * #failures) + (1*time of finding)
  not finding at least as important as finding

we need

- algorithm for
  - fast detection of no solution/not finding
  - fast finding of feasible solution
- strategy to
  - select tradeoffs
  - choose time spent
  - allow for detection of why no solution found (difficult)
  - good redesign for next schedule attempt
- designer support

most current algorithms concentrate on finding solution only

Directions

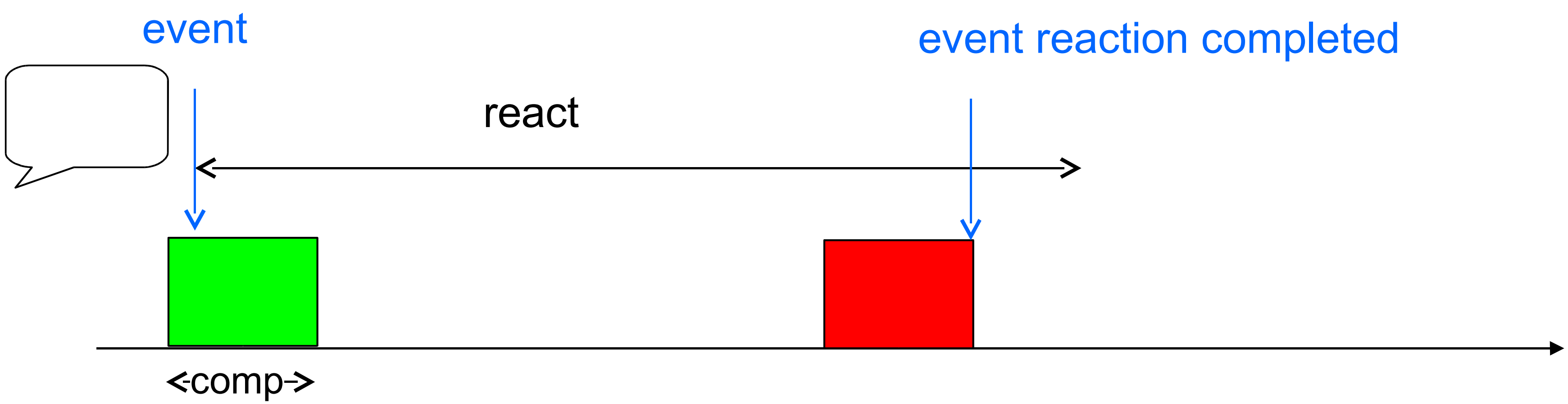
How to construct a schedule?

- simple solution: use online scheduling, e.g., EDF
  - still better than online - can backtrack or redesign
  - better utilization because resource conflicts are known, don't need to assume worst case
  - testing
  - etc.
- search
  - popular
  - easy to change constraints
  - easy algorithm
  - problems with feedback problem - source in search tree

- genetic algorithms
  e.g., simulated annealing
  - simple
  - does not get stuck easily with hard sub problems
  - can handle large task sets
  - difficulties with complex constraints
  - good for allocation of tasks to nodes in distributed system
- “by hand”
  - sometimes really fully by hand
  - with support
    - resolve difficult parts by hands
    - extend existing schedules
    - place some tasks by hand

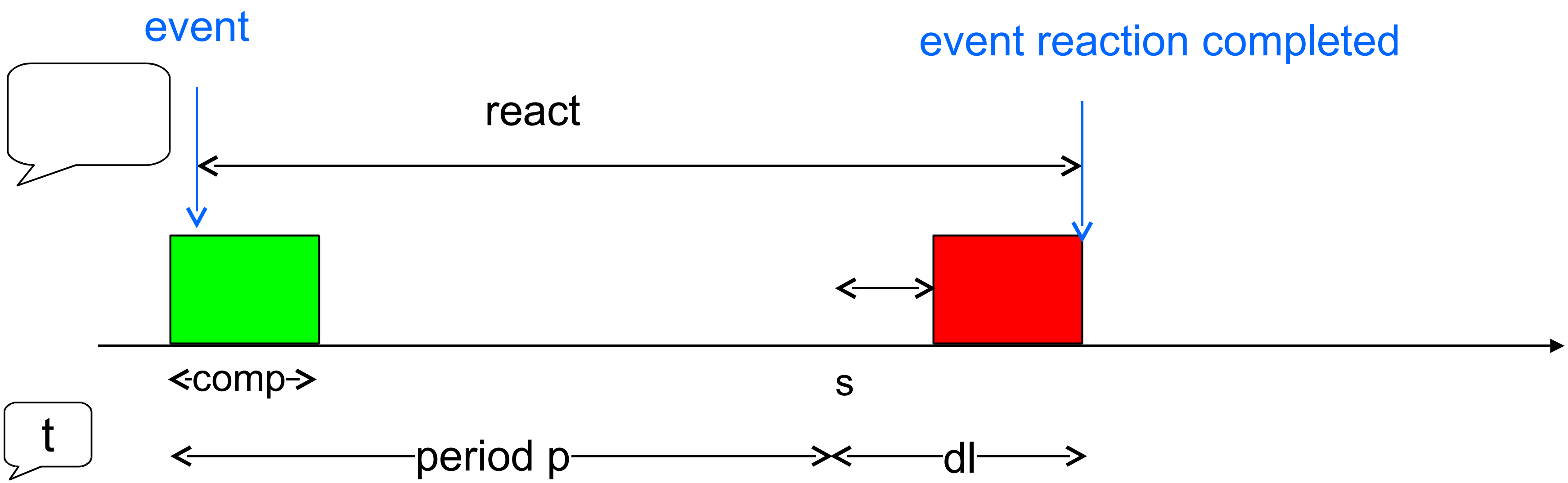
Making a periodic world

- “naturally periodic”, e.g., control, sampling
- aperiodic tasks, i.e., without any restriction on arrival
  no way
- sporadics
  transform into *pseudo periodic tasks*
  assumptions about *events*
  - maximum rate of change, *minimum inter arrival interval, mint*
  - maximum delay of reaction, *react*
  - computation time, *comp*
- determine period and deadline
- have to ensure that
  1. reaction is not late
  2. no event missed



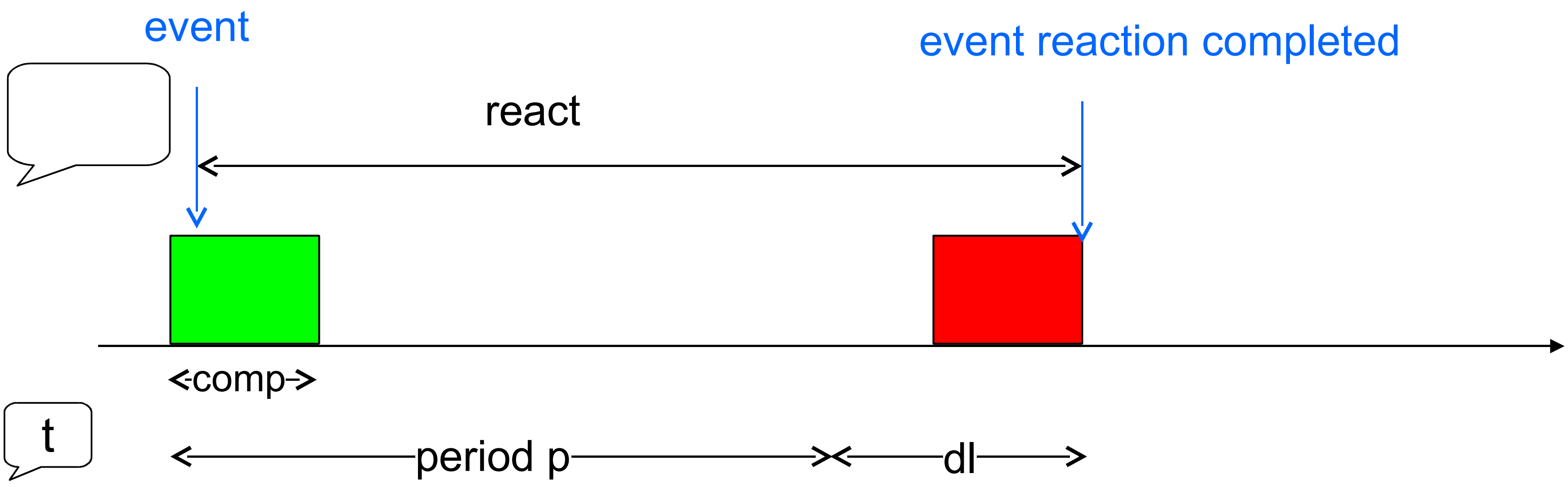
- worst case:
  event happens right after task start - misses data just by

  event gets reacted by task only at next instance invocation

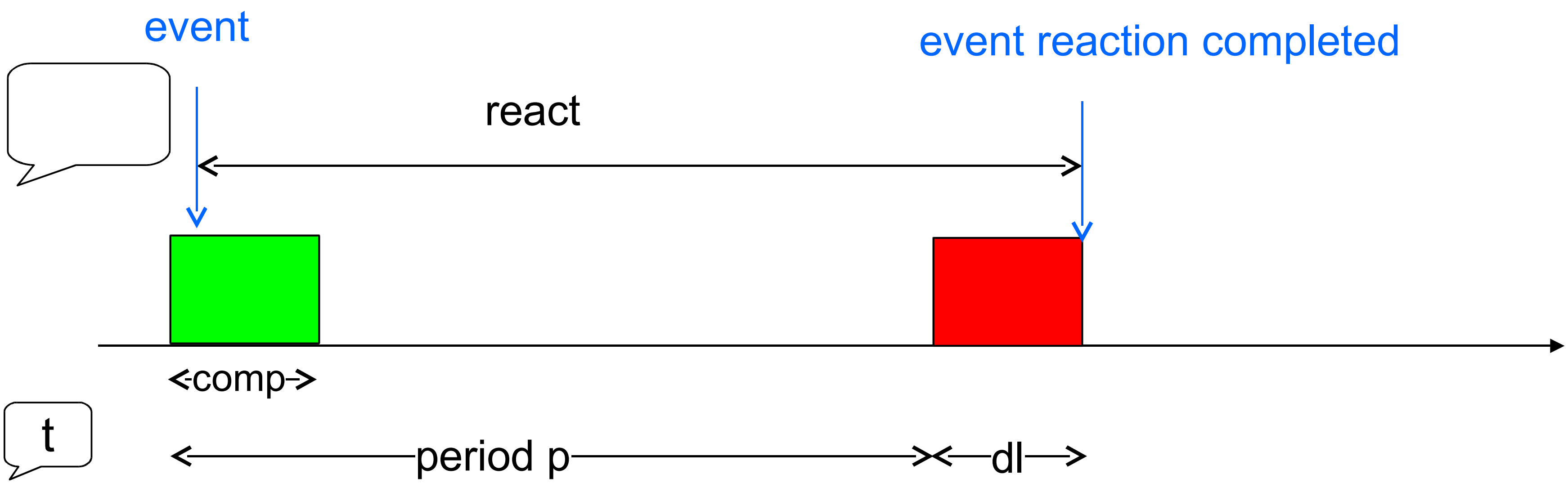


- deadline
  dl=comp+s, s 0
- next instance completes no later than *react* after event
  - event starts at t +
  - reaction finishes at t + p + dl
  - t + p + dl - t -   react

p + dl react +   or p + comp + s react +



- maximum value for p - not react too late
  p < react +   - dl or p < react +   - comp - s
- maximum value for p - not miss event
  p < mint



- assume dl=comp; s=0

$$p \quad \begin{matrix}\text{react - comp}\\ \text{mint}\end{matrix}$$

- Utilization:

$$U_0 \quad \frac{comp}{p} \quad \frac{comp}{react \quad comp}$$

- assume dl=comp+s; s>0

$$U_s \quad \frac{comp}{p} \quad \frac{comp}{react \quad comp \quad s}$$

- $U_0 < U_s$!

- period and deadline dependent on each other
- tradeoff
  - large period:
    - low utilization demand
    - tight deadline - schedulability problems
  - small period:
    - relaxed deadline
    - high utilization demand

- if events are *rare,* but urgent when they occur transformation inefficient, high utilization demands
  e.g.,
  mint=1000*comp; react=2*comp:
  p < react + - comp = comp +

$$U \quad \frac{comp}{comp} \quad 1$$

- monopolization of CPU

- actual need to handle event without pseudo periodic transformation

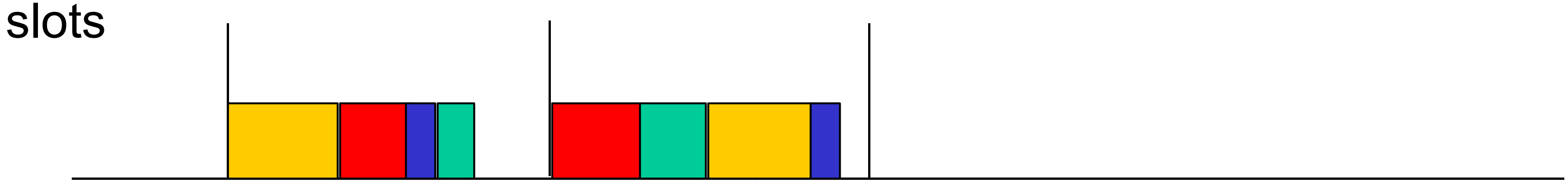
$$U \quad \frac{comp}{1000 * comp} \quad 0.001$$

Off-line Scheduling and the Real World

- Many algorithms assume tasks, messages, slots, constant operating system overhead
- real-world demands
  - interrupts
  - threads, chains
  - micro kernel OS
    - system threads
    - task ensembles for tasks, e.g., message transmission
    - depending on scheduling and allocation
    - dynamic creation of threads
- do not fit into off-line schedule in straightforward way

Threads

- threads are shorter than granularity of slots
- better utilization of slots
- scheduling/dispatching happens not only at slot boundaries

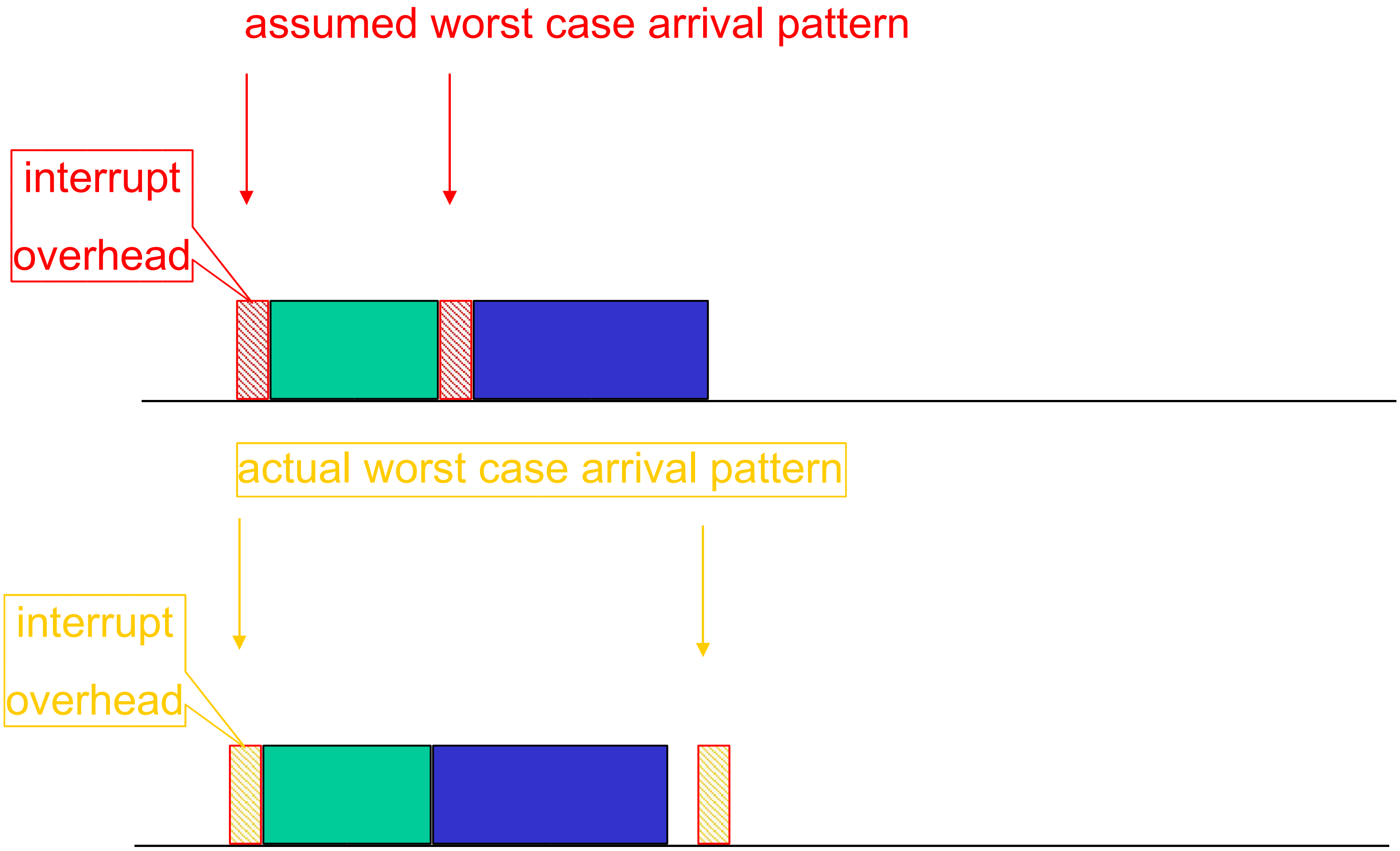


- scheduler needs to construct chains as well
- offline scheduler does “micro scheduling”, e.g., thread cumulating within slot
- backtracking, heuristic etc only at slot boundaries
- not optimal, but tractable

Interrupts

- interrupts have to be considered
- cannot
  - ignore them - too much time demand
  - handle them as tasks/threads -
    too high overhead, too long response times
  - have to account for in analysis during schedule construction
  - minimum inter arrival time - maximum overhead
- naïve approach
  - assume each task can be hit by a worst case arrival of interrupts
  - ala exact analysis
  - very high overhead

- if task is shorter than minimum inter arrival time interrupt overhead is considered too often for two consecutive tasks



Time triggered vs. event triggered

Who is doing what, when?

- Run-time dispatching is performed according to a ***set of rules***.
- Off-line analysis and testing has to ensure that the provided rules for the run-time dispatcher are correct:
  - when the dispatcher takes scheduling decisions according to the given rules, all timing constraints are kept.
  - ***off-line guarantees***

TT:

- offline scheduling
- rules for runtime dispatcher expressed as scheduling table

ET:

- online scheduling, priority driven
- rules applied at runtime, e.g.,
  - earliest deadline first (dynamic priority)
  - fixed priority

Properties – Time Triggered

activities initiated at predefined points in time
everything planned before system is deployed

- offline constructed scheduling table
- runtime dispatcher executes decision in table

☺ deterministic – exact behavior known beforehand
☺ test space dramatically reduced
☺ complex demands: distributed, jitter, engineering practice, …
☺ low runtime overhead – table

☹ inflexible – can only handle what is completely known before
☹ inefficient – based on worst base

widely used in safety critical, e.g., automotive, avionics

Properties – Event Triggered

- online scheduling, priority driven
- event activates scheduler which takes decision
- priority rules + test
  - earliest deadline first (dynamic priority)
  - fixed priority

☺ flexible – not completely known activities can be added easily
☺ widely used

☹ only simple demands
☹ high runtime overhead for semaphores, blocking, ...
☹ non determinism
☹ high testing efforts - keeps deadlines, but cannot determine when exactly

TT, ET totally different?

- dispatcher same basic operation, executing rules
  - TT: rules as scheduling table
  - ET: rules as functions

- online scheduling can provide more flexibility, but *no magic:*
  - what is not exactly known before run-time cannot be guaranteed offline, independent of the used scheduling strategy.

- TT assumes periodic world, ET does not
  BUT: for offline guarantees, ET assumes periodic as well
  (to ease analysis)

- want both
  - integrated offline and online scheduling
  - offline:
    - basic guarantees
    - realistic constraints
  - online:
    - for efficient resource usage and flexibility
- methods for combined use exist