Course 02224

Scheduling

Further Notions

Hans Henrik Løvengreen

DTU Compute, Technical University of Denmark

Outline

Basic Scheduling Theory Summary

Exectution Model

- N independent, periodic *tasks*
- Task parameters: $(T_i, C_i, D_i, \dots)$

Fixed Priority Scheduling

- Utilization check: $\sum_{i=1}^{N} \frac{C_i}{T_i} < N(2^{1/N} - 1)$
- Response time analysis: $R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$

Dynamic Priority Scheduling

- Typical: *Earliest Deadline First* (up to $U = 100\%$)

Deadline Monotonic Priority Assignment

- A *deadline monotonic* priority assignment satisfies:

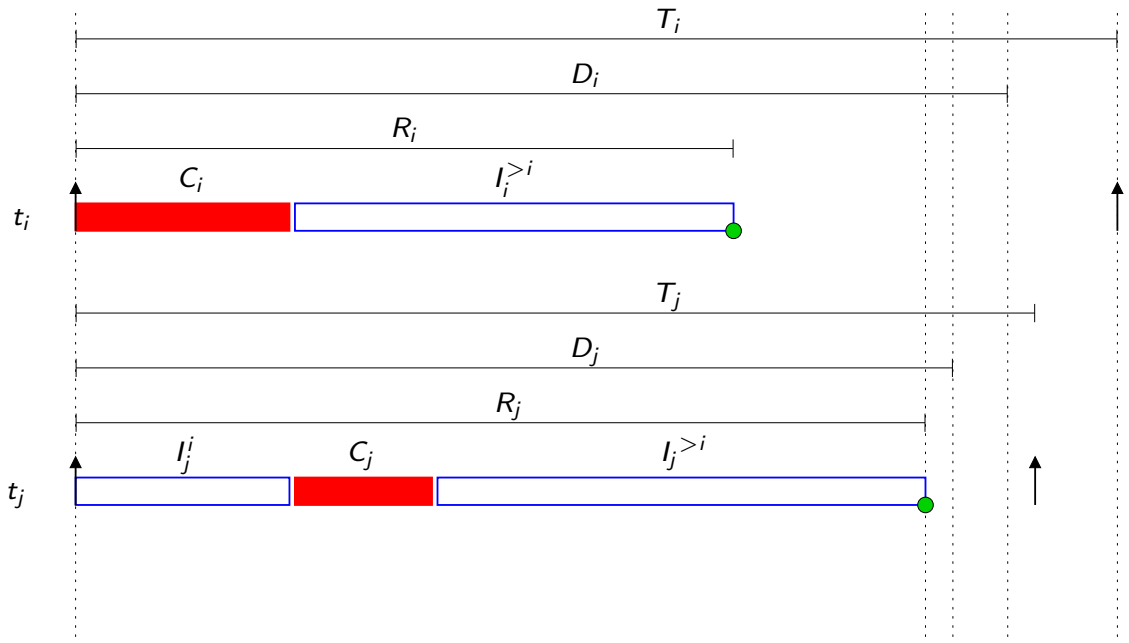
$$D_i < D_j \Rightarrow P_i > P_j$$

- Deadline monotonic priority assignment is optimal for independent processes

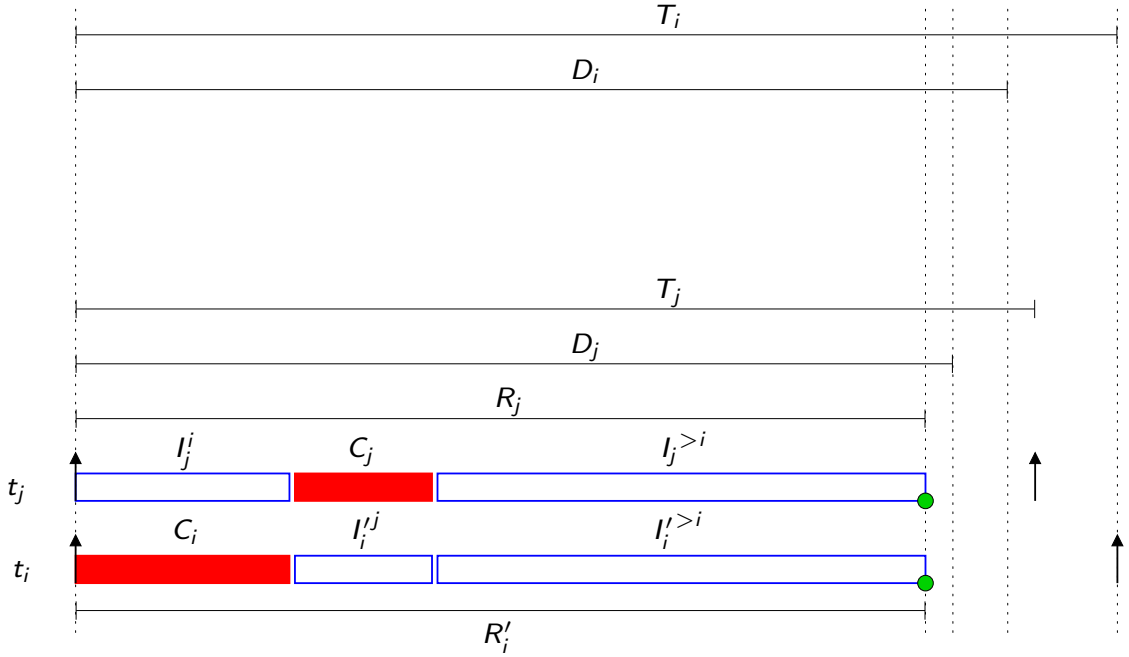
Proof (sketch)

- Given a feasible priority assignment W
- Swap any two adjacent tasks with $P_i > P_j$ but $D_i > D_j$
- When none left, assignment is feasible and deadline monotonic
- Corrollary: RMA is optimal for $D_i = T_i$

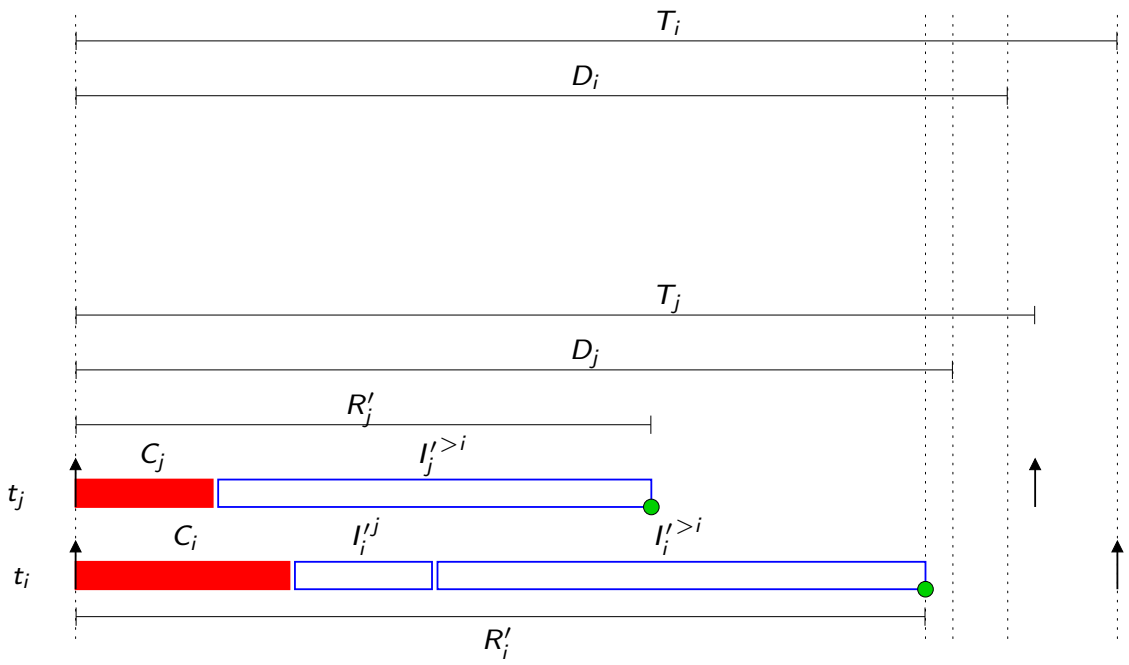
Optimality Proof



Optimality Proof



Optimality Proof



Outline

Task Dependencies

- Basic scheduling theory assumes that tasks are independent.
- In practice tasks cooperate or compete:
  - Tasks may share data and other resources
  - Tasks may depend on conditions set by other tasks

Task Interaction by Sharing

Assumptions

- Tasks interact by shared variables protected by *critical regions* (eg. monitors).
- Access to critical regions is strictly by priority.

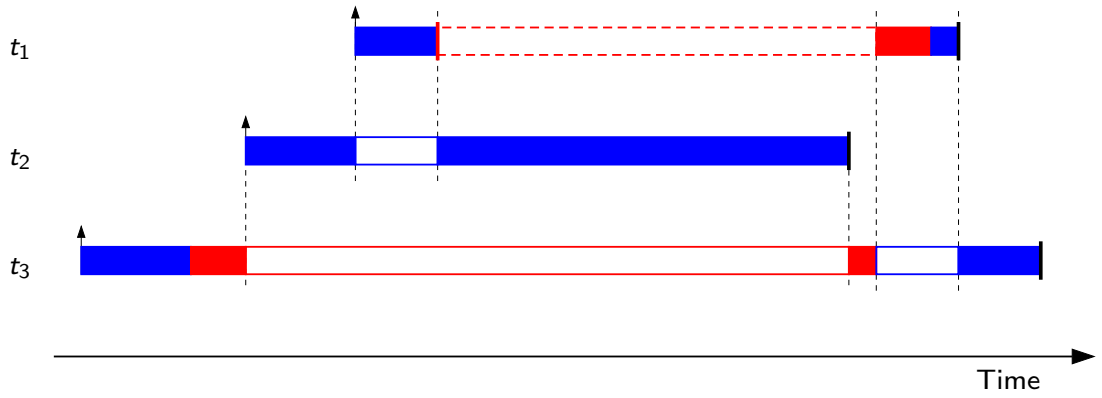
Impact of interaction

- A high priority task may be *blocked* by requesting access to a critical region held by a lower priority task.
- Let B_i be the *maximum blocking time* task i may be blocked by lower priority tasks.
- Blocking time contributes to the response time:

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

Priority Inversion

Problem



- *Priority inversion* occurs when a high priority task is indirectly blocked by a runnable low priority tasks.
- Famous example: Mars Pathfinder Base station (1997).

Priority Inheritance Schemes

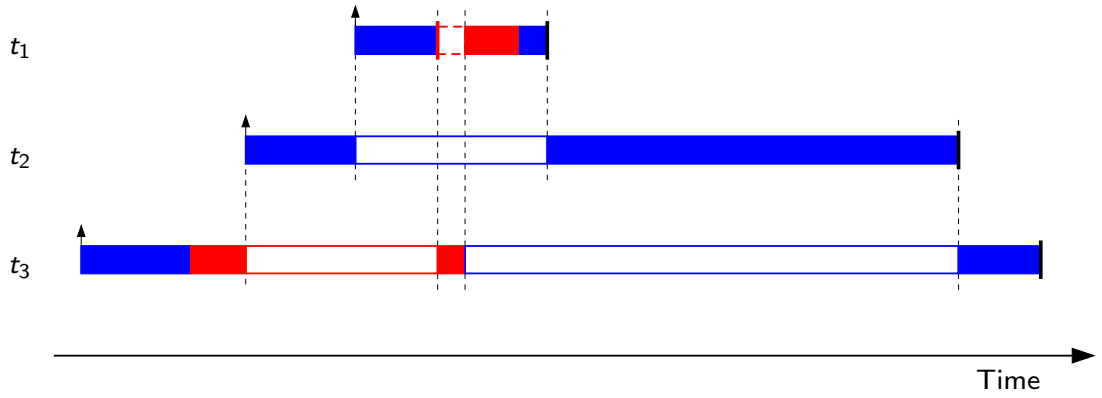
Priority Inheritance

- Attempt to eliminate blocking and priority inversion by priority propagation.
- Standard algorithm: Priority *boosted* when blocking.
- Still allows for transitive and accumulated blocking.
- *Priority ceiling* protocols reduces blocking even further.

Priority Inheritance Protocol

- Each task has a static *base priority*.
- The *dynamic priority* of a task is raised whenever a task blocks a task with a higher (dynamic) priority.

With inheritance:



Immediate Priority Ceiling Protocol

- Each task has a static *base priority*.
- Each critical region has a static *ceiling priority* being the maximum base priority of any task using it ($+1$).
- The *dynamic priority* of a task is the maximum of its base priority and the ceiling priorities of any critical regions it holds.

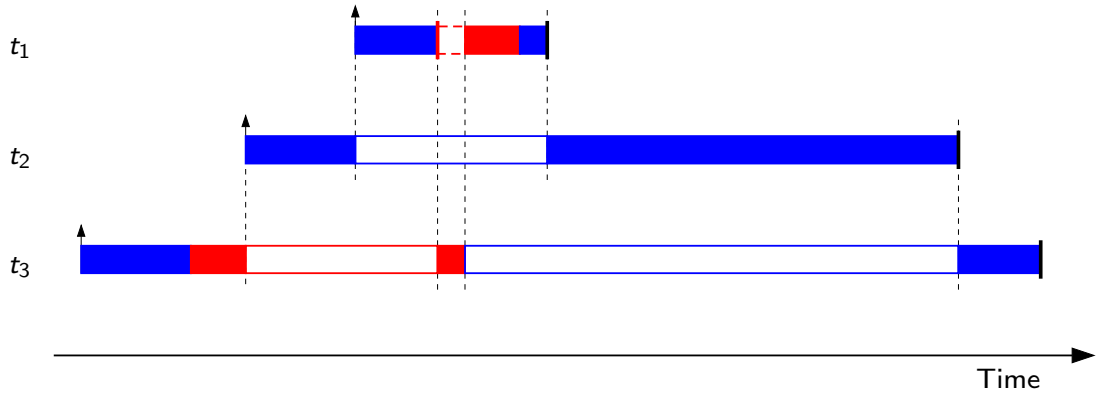
Properties

- A task can be blocked by at most by one lower priority task.
- Transitive blocking is prevented.
- Mutual exclusion in ensured (on mono-processors).
- Deadlock is prevented (on mono-processors).

Immediate Priority Ceiling Protocol

Example

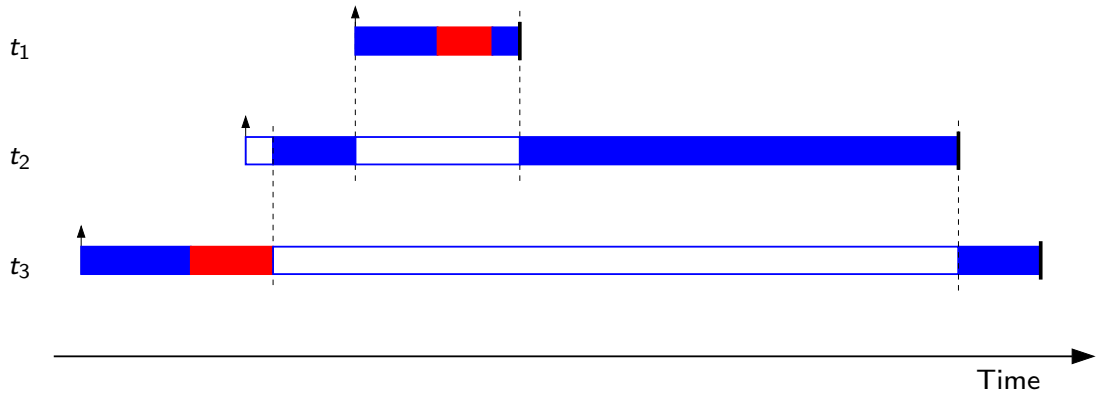
- Basic inheritance:



Immediate Priority Ceiling Protocol

Example

- Immediate priority ceiling:



Original Priority Ceiling Protocol

- Sha, Rajkumar & Lehoczky 1990.

Principle

- Each task has a static *base priority*.
- Each critical region has a static *ceiling priority* being the maximum base priority of any task using it $(+1)$.
- The *dynamic priority* of a task t is the maximum of its base priority and the priority of any task that is blocked by t.
- A task can enter a critical region if its dynamic priority is greater than the ceiling priority of any region occupied by other tasks.
  Otherwise the task is said to be *blocked* by the task occupying the region with the highest ceiling priority.

Original Priority Ceiling Protocol II

Compared to Immediate Ceiling Protocol:

- Same worst case performance.
- Often better average response times.
- Same freedom from deadlock.
- Much more complex to implement.

Outline

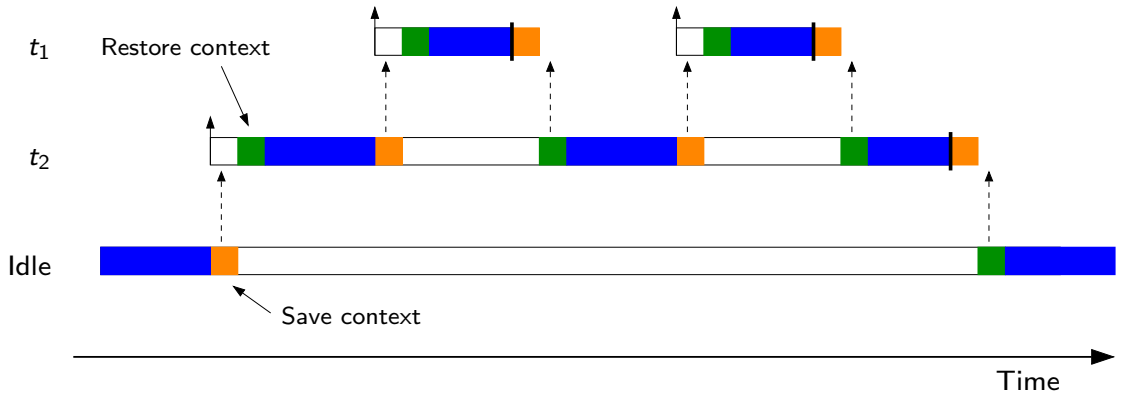
Jitter

- Derivations from periodical behaviour is called *jitter*.
- The *release jitter* J_i may be seen as a potential delay of the actual release wrt. ideal release.
- May be caused by *timer granularity*.
- Release jitter may be incorporated in the response calculation:

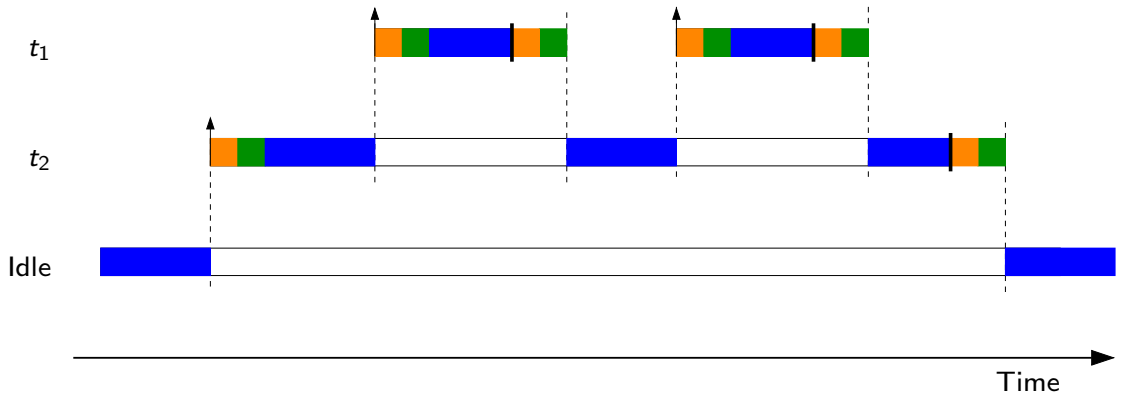
$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i + J_j}{T_j} \right\rceil C_j$$

- Response time wrt. ideal release is given by: $R_i^* = R_i + J_i$

Overhead — Context Switching



Overhead — Context Switching



- $C_{\mathrm{CS}} = C_{save} + C_{restore}$
- Use $C_i^* \triangleq C_i + 2C_{\mathrm{CS}}$ instead of C_i

Interrupts

Interrupts

- Timer interrupts typically release periodic tasks
- Other interrupts typically release aperiodic tasks
- Interrupt service routines may be considered high priority tasks
- Nested interrupt handling corresponds to preemption

Task Precedence

- Tasks may have *phases* with different requirements
- Tasks may have to *synchronize and communicate*
- Tasks may *release* other tasks

Subtasks

- A *subtask* represents a unit of computation
- Subtasks may have more or less elaborate *precedence constraints*
- Different priorities may be assigned to related subtasks
- Standard analysis can accomodate some constraints
- Often *ad hoc* analysis necessary

Other RTA Extensions

The FPS response time analysis may incorporate:

- Arbitrary deadlines ($D_i > T_i$)
- Cooperative scheduling
- Fault tolerance
- Offsets

Outline

Multi-processor Scheduling

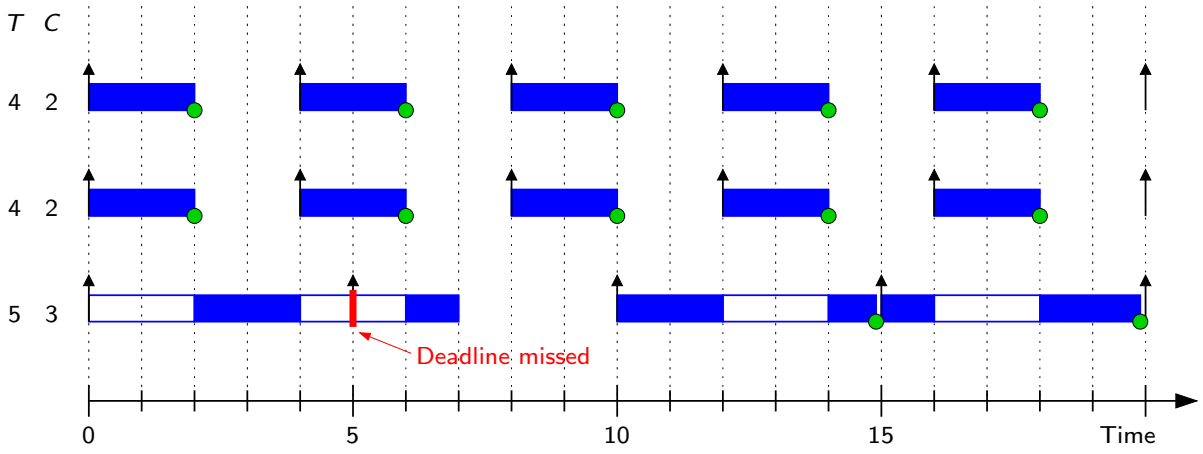
- Classical theory adresses only a single CPU
- Multi-processors are becoming standard (SMP)

Issues of Multi-processor Scheduling

- Mono-processor policies like RMA and EDF no longer optimal
- Scheduling *anomalies* may occur.
- Global scheduling in general hard to analyze

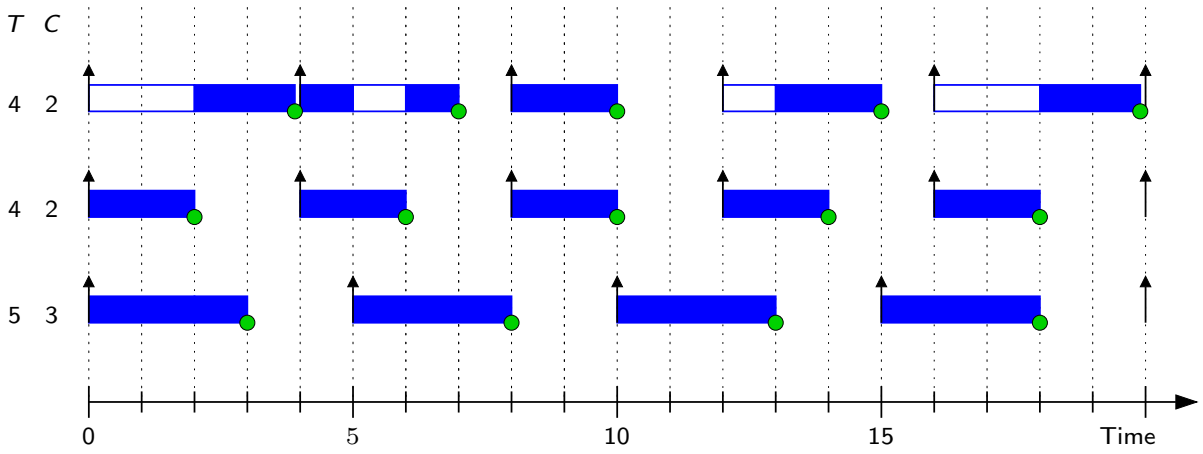
Multi-processor Scheduling — 2 CPUs

- RMA



Multi-processor Scheduling — 2 CPUs

- Inverse RMA



Multi-processor Scheduling

- Classical theory adresses only a single CPU
- Multi-processors (multi-cores) have become standard
- Allow for *global scheduling*, aka. symmetrical multiprocessing (SMP)

Issues of Multi-processor Scheduling

- Mono-processor policies like RMA and EDF no longer optimal
- Scheduling *anomalies* may occur
- Global scheduling in general hard to analyze
- Conclusion: SMP not suited for hard real-time
- Typical alternative approach — *partitioned scheduling*:
  - Assign tasks to processors
  - Use local scheduling on each processing node

Multi-level Scheduling

- Many larger systems have a mixture of hard, firm and soft tasks
- Can be handled by running multiple schedulers

Some Approaches

- Separate *bandwidth preserving server* to handle aperiodic tasks
- Running soft real-time tasks in low-priority task (RT Linux)
- General two-level scheduling
- Processor *virtualization*

Outline

Worst-case Execution Times

- C_i is essential for schedulability analysis!
- Can be *measured* or *calculated*
- Both approaches are tedious and error-prone
- Analysis tools are emerging
- Obstacles with modern processors:
  - Caching
  - Pipelining
  - Multiprocessing
- Safe approach: Use a simple processor anno 1980.
- Research: Design processor for low WCET

Summary of Scheduling Theory

- Scheduling theory was initiated in the 1970'ies
  — but forgotten again until the 1990'ies
- Theory focusses on *hard real-time* properties (worst-case)
- Not much knowledge or usage within profession
- UML-based analysis tools may change best practice
- Limited to simple periodic tasks and deadlines.
- A *real-time operating system* should provide:

  fixed priorities + inheritance
- For *soft real-time* systems, a lot of performance theory exists
- Focus on average *end-to-end* properties and *quality of service*
- General OS try to improve their (soft) real-time properties

Outline

Tools for Schedulability Analysis

Academic Tools

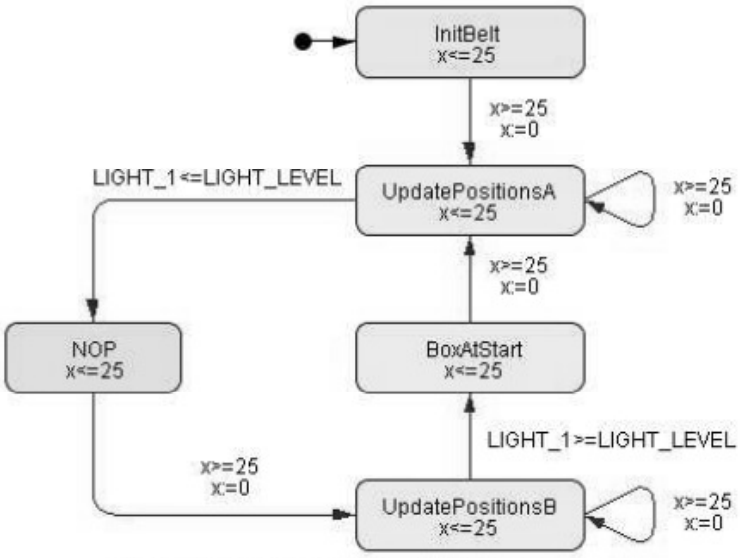
- Times from the UPPAAL-group
- MAST from Cantabria University (Spain)
- Cheddar from University of Brest (France)

Commercial Tools

- TimesWiz from TimeSys
- RAPID RMA from TriPac
- Integrated with UML-based IDEs like Rational Rose and Rhapsody
- May use the UML 2.0 *schedulability profile*, or the later MARTE (Modelling and Analysis of Real-Time Embeddded Systems)

Times: Extended Timed Automata

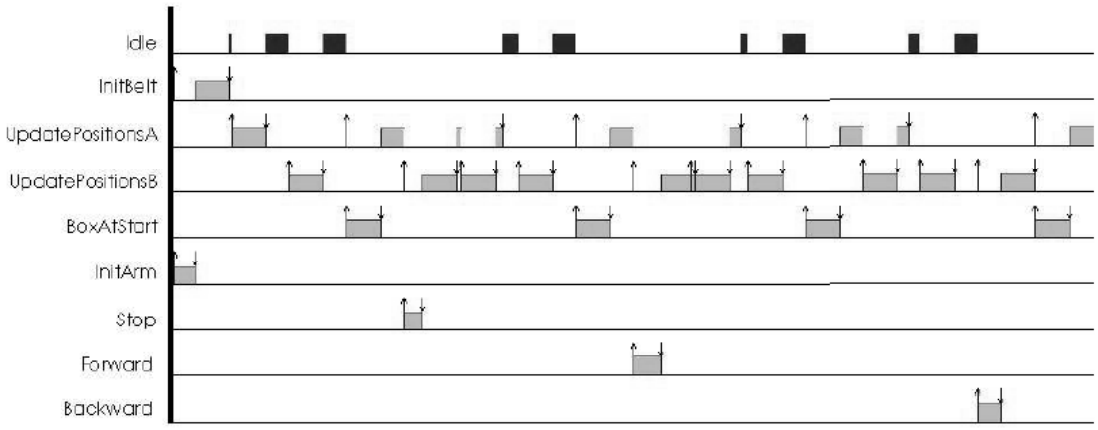
- Task are characterized by the parameters $C, D, \ldots$
- They may be released by arbitrary *timed automata*:



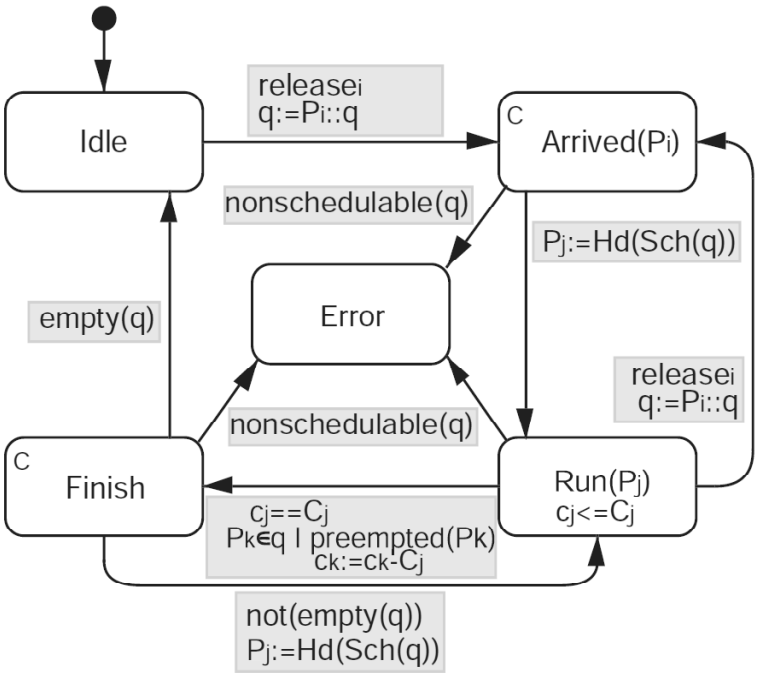
(a) The Feed Belt control automaton

Times: Generated Schedule

- Released tasks interact with a *scheduler automaton* using a given policy:

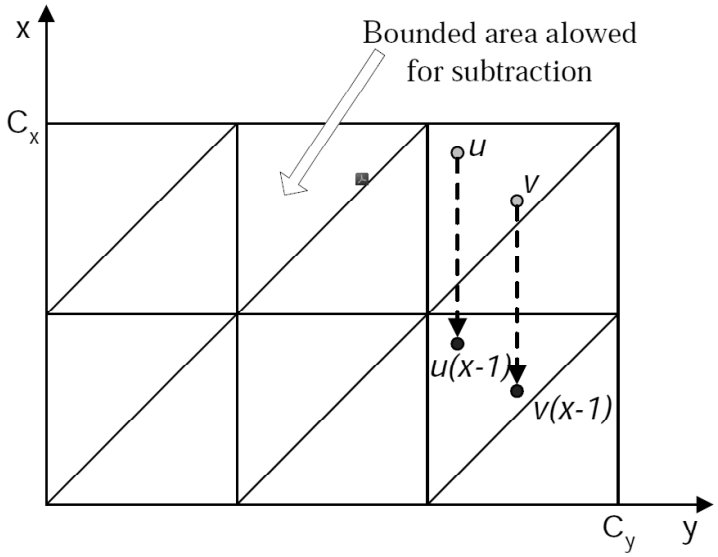


Times Scheduler Automaton



Region Subtraction

- Each tasks has an clock c_i of accumulated computation time
- For a preemption of C_j units, clock adjustment preserves regions:



Scheduling Verification using UPPAAL with Stopwatches

- UPPAAL 4.1 introduces *stopwatches*
- In a location a clock x either runs $x' = 1$ or is stopped $x' = 0$
- By default, all clocks run in a location
- Stopwathces render the verification *undecidable*
- Using *overapproximation* properties may still be verified
- Stopwatches may naturally model *preemption* of tasks
- A general scheme utilizing this is described in:

  David, Illum, Larsen, Skou: *Model-based Framework for Schedulability Analysis Using Uppaal 4.1*, 2009

- Exists as SchedulingFramework.xml in the demo folder

Cheddar

- Developed at the University of Brest 2002–
- Available at `beru.univ-brest.fr/cheddar`
- Model: Multiple *tasks* scheduled on *processors* with multiple *cores*
- May do some static analysis, but is basically a *simulation tool*
- Supports the system definition langauge AADL
- Implemented in Ada (and supports Ada RT notions)
- Integrated in both UML-based and AADL-based tools

Cheddar Example: Multi-processing