Technische Universität Dresden

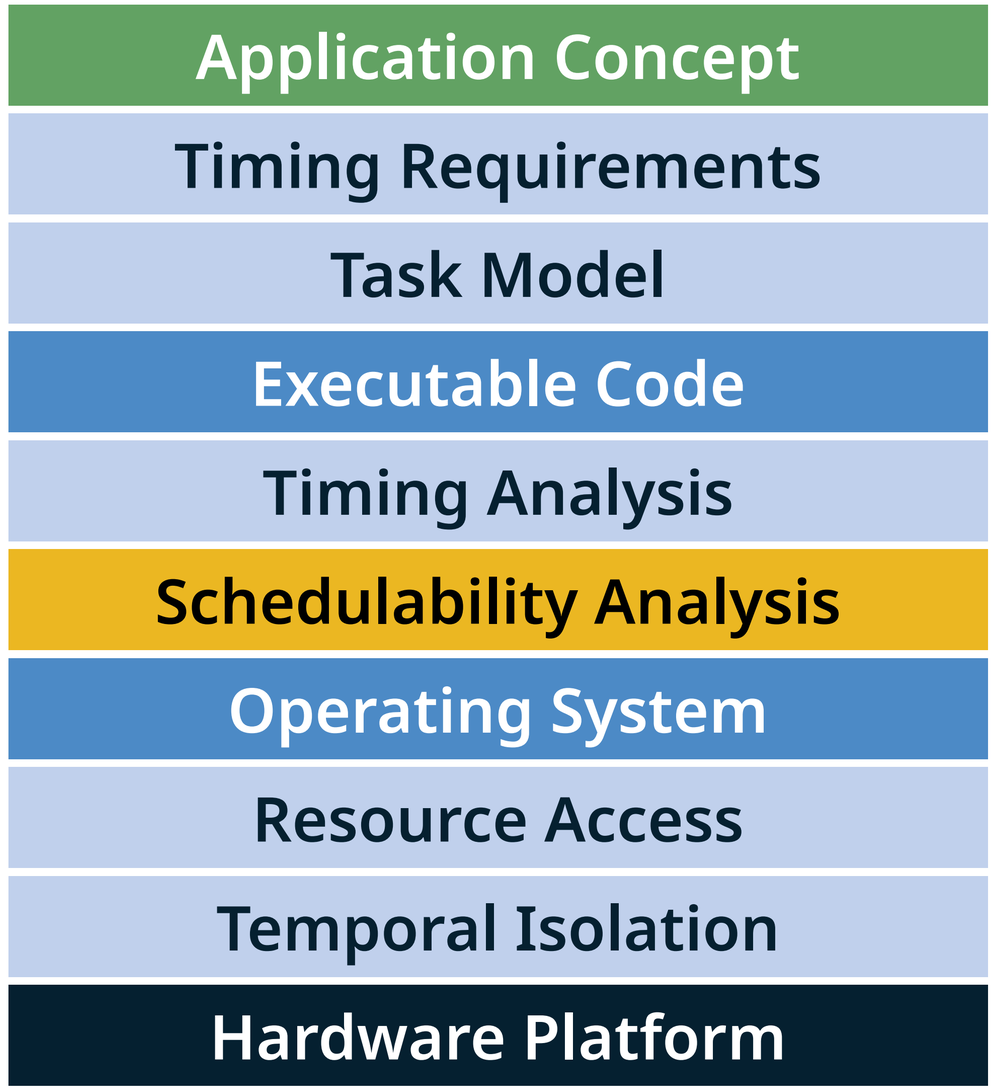
barkhausen institut

Event-Driven Systems

Real-Time Systems

Michael Roitzsch

Real-Time Systems Technology Stack



Restrictions of Time-Driven Systems

We want to relax some restrictive assumptions of time-driven systems:

- fixed number of real-time tasks
  → number of real-time and non real-time tasks can vary
- fixed inter-release times
  → minimum inter-release times
- all task parameters fairly well-known a priori
  → overload, schedule non-RT in the background, ...

Principles



Important Properties

- scheduling decisions are triggered by events (not time instants)
- events are release, completion, blocking, unblocking of jobs
- event triggers: scheduler calls, interrupts, timers, ...
- scheduling decisions are made online
- scheduling must therefore be simple
- admission is online or offline
- *work-conserving* schedulers never leave a resource idle intentionally

Workflow



At Admission Time

- select scheduler (may depend on the OS)
- check if feasible schedule exists for the selected scheduler
- assign jobs a value as a simple selection criterion: priorities

Scheduling / Dispatching

- at event, select highest prioritized job

Comparing Schedulers



How good are schedulers?

- shorter response times (especially for the most important task)
- higher utilization of resources
- more task sets lead to a feasible schedule

Optimality of Schedulers

- A scheduling method X is called *optimal in a class of scheduling methods,* if X produces a feasible schedule whenever there exists a scheduling method Y in this class that produces a feasible schedule.
- X is called *optimal,* if X produces a feasible schedule whenever there exists such a schedule (no matter which method produced it).

Technische Universität Dresden

barkhausen institut

Classical Algorithms: RMS & EDF

Assumptions for Algorithms

Set of **periodic tasks** with these properties:

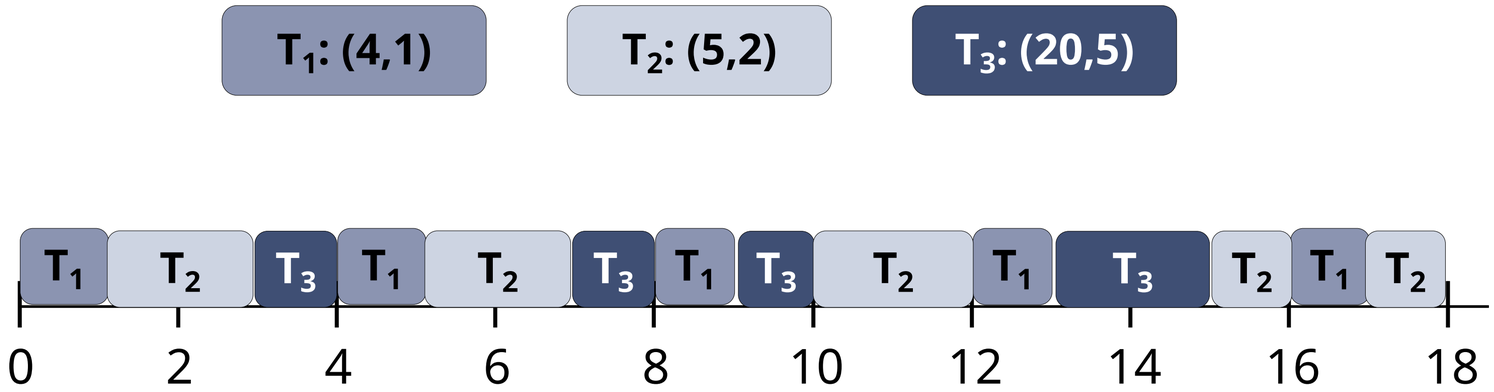
- tasks are independent
- one processor
- preemptable, context-switch overhead is negligibly small
- known period = minimum inter-release separation
  (release times are not fixed but at least one period apart)

Since tasks are independent, tasks can be added (if admitted) and deleted at any time without causing deadline misses.

Rate Monotonic Scheduling

Fixed Priority Scheduling

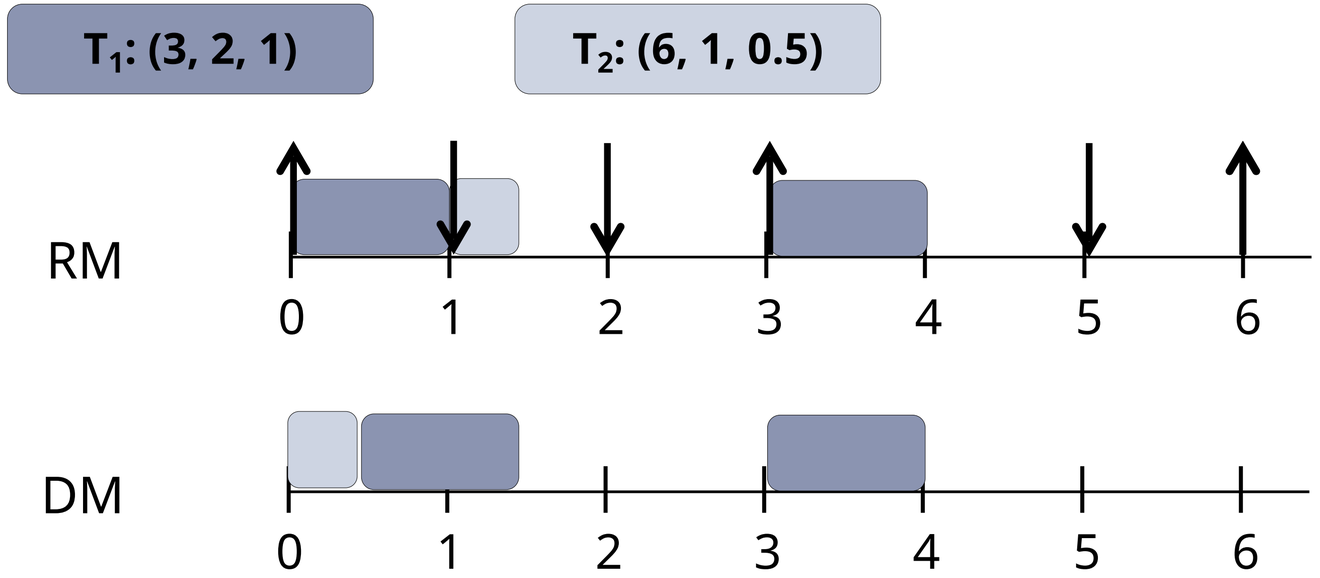
- the shorter the period the higher the priority (rate: inverse of period)
- example: (p, c); deadline = period



Deadline Monotonic Scheduling

Fixed Priority Scheduling

- the shorter the relative deadline the higher the priority
- example: (p, d, c)



Conclusion (no proof): if $d \leq p$ for all tasks, RM not optimal, but DM optimal

Optimality of Fixed Priority Schedulers



T: set of periodic tasks, independent, preemptable, single CPU

Rate Monotonic Scheduling (RMS)

- relative deadlines = periods
- if there is any feasible fixed priority schedule for T,
  then Rate Monotonic priority assignment results in a feasible schedule

Deadline Monotonic Scheduling (DMS)

- relative deadlines ≤ periods, in phase
- if there is any feasible fixed priority schedule for T,
  then Deadline Monotonic priority assignment results in a feasible schedule

Static and Dynamic Priorities

If no new tasks arrive:

- task-static priorities / fixed priorities:
  task T does not change its priority, i.e. all jobs of T have same fixed priority
- job-static priorities:
  jobs do not change their priorities
- job-dynamic priorities:
  jobs change their priorities

Careful: job-static priorities is already a dynamic-priority system

Earliest Deadline First

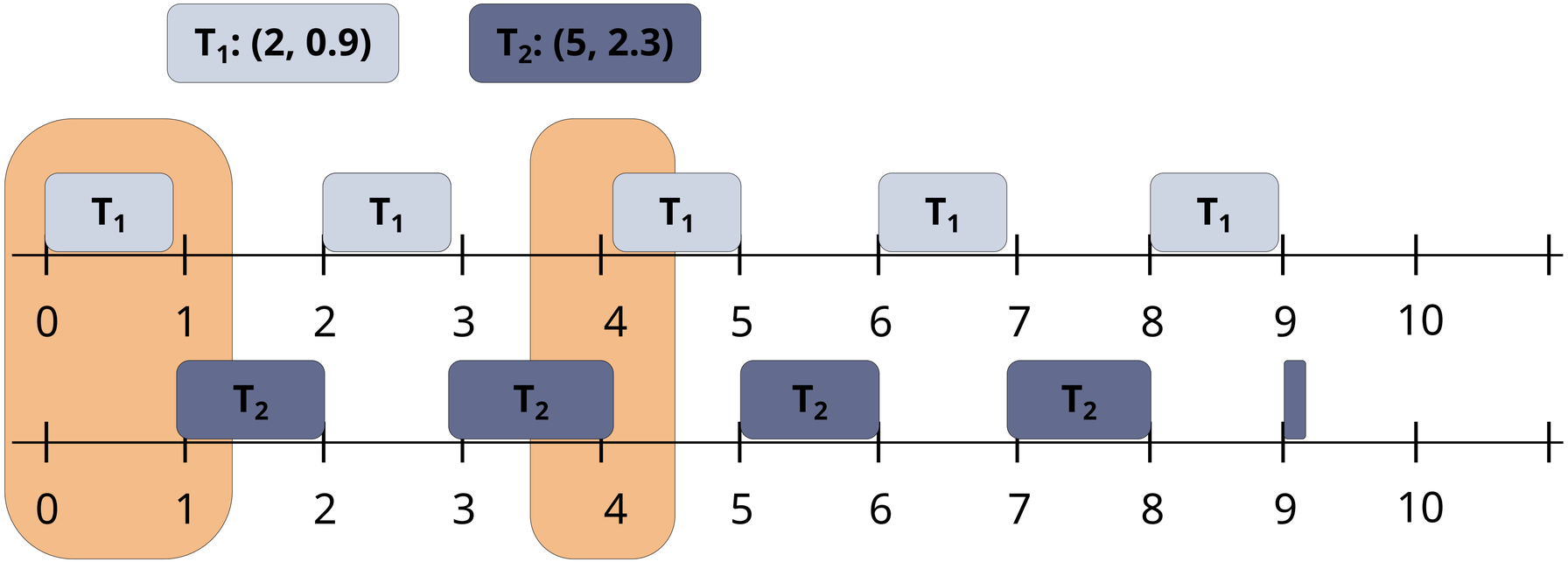
T: set of periodic tasks, independent, preemptable, single CPU

Scheduling decision at runtime:
“the earlier the deadline the higher the priority”

- EDF is optimal
- if there is any feasible schedule for T,
  then Earliest Deadline First scheduling also produces a feasible schedule

Earliest Deadline First

priority assignment fixed per job, but dynamic at task level:
the closer the absolute deadline of a job at its release, the higher the priority



Technische Universität Dresden

barkhausen institut

Schedulability Testing

Admission Based on Utilization



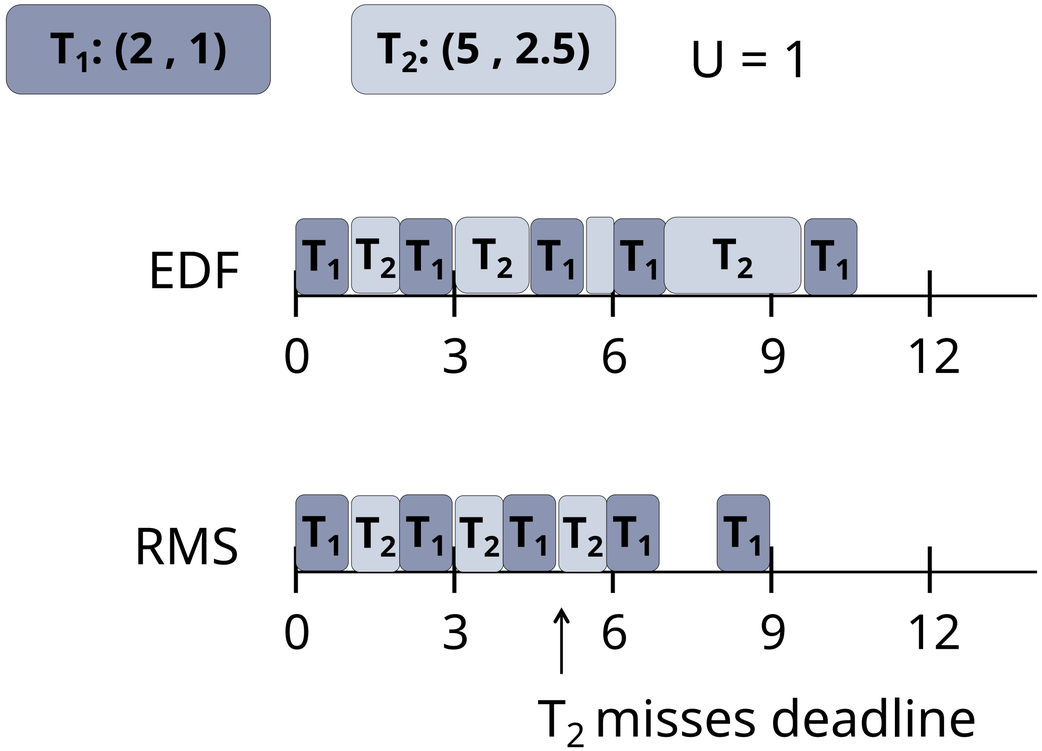
Utilization U

- a task (p, c) requires c/p of the capacity of a processor
- any scheduler can admit at most up to full capacity on m processors:
  For a task set $T_1 \ldots T_n$: $\sum c_i/p_i \leq m$ is a necessary but not sufficient condition

Schedulable Utilization / Utilization Bound

- can we establish a maximum utilization bound SU such that:
  $T_1 \ldots T_n$: $\sum c_i/p_i \leq SU$ is sufficient?
- depends on the scheduling algorithm
- higher SU are better, because the test is less pessimistic
- test is not exact: utilizations greater SU *may be* schedulable

Schedulable Utilization: RMS vs. EDF



RMS not optimal in general → static priorities are ‘less powerful’ than dynamic

Schedulable Utilization Results



Periodic Tasks

independent, preemptable, deadline = period, single processor

n number of tasks

EDF: SU = 1

RMS: SU = $n\,(2^{1/n} - 1)$

for $n \to \infty$: SU $\to \ln(2)$

RMS with harmonic periods (periods are integer multiples of each other):

SU = 1

Alternative Test for Fixed Priority Schedulers

for periodic task sets with $d_i \leq p_i$

Response Time Analysis / Critical Instant Analysis

- critical instant for task T_i:
  release of jobs such that they have the maximum response time
- 1 CPU, preemptable, independent:
  critical instant occurs when all tasks are released simultaneously
- it is sufficient to check schedulability for the critical instant until the longest period in the task set

Fixed Priority Schedulability and Blocking



- T_i may have to wait for non-preemptable, lower priority task
- insight: blocking-related waiting can only be caused by lower priority tasks
- b_i: longest non-preemptable portion of all lower priority jobs
- schedulable utilization for a task set $T_1 \ldots T_n$: analyse iteratively $i = 1 \ldots n$

  $U(i) = c_1/p_1 + c_2/p_2 + \ldots + c_i/p_i$

  $U(i) + b_i/p_i \leq SU(i)$

Non-Negligible Context Switch Time

- For job-level fixed priority schedulers:
  each job preempts at most one other job
- 2 context switches per job:
  - at release (when it preempts a lower-priority job)
  - at completion (when switching back to the lower-priority job)
- include context switch overhead in each job's WCET c_i:
  $c_i' = c_i + 2 \times$ context switch time

Sustainability Results for Schedulability Tests



- remember sustainability:
  if any task set is considered schedulable by a specific schedulability test, then it remains schedulable if it behaves “better” than described by its task parameters
- utilization-based schedulability test for RMS **is sustainable** with respect to execution time requirements, deadlines, and periods
- response time analysis of fixed priority preemptive task sets with independent tasks **is sustainable** with respect to execution time requirements, relative deadlines and periods
- any sufficient EDF-schedulability test for periodic task systems on preemptive uniprocessors **is sustainable** with respect to execution time requirements and relative deadlines

Beyond Sustainability



Predictable Execution

- given a set of periodic tasks with *known minimal and maximal* execution times and a scheduling algorithm
- a schedule produced by the scheduler when the execution time of each job has its minimum (maximum) value is called a *minimum (maximum) schedule*
- an execution is called *predictable,* if for each actual schedule the start and completion times for each job are bound by the respective times in the minimum and maximum schedules
- the execution of every job in a set of independent, preemptable jobs with fixed release times is *predictable* when scheduled in a priority-driven manner on a single processor

Technische Universität Dresden

barkhausen institut

More Dynamic-Priority Scheduling

Scheduling Based on Remaining Slack Time



Slack Time / Laxity

- time to deadline – remaining execution time
- d – x – t

  d absolute deadline

  x remaining execution time of the job

  t current time

(Least / Minimum) (Slack Time / Laxity) First

- priority dynamic per job: lower slack → higher priority
- strict version (see next slide) is optimal

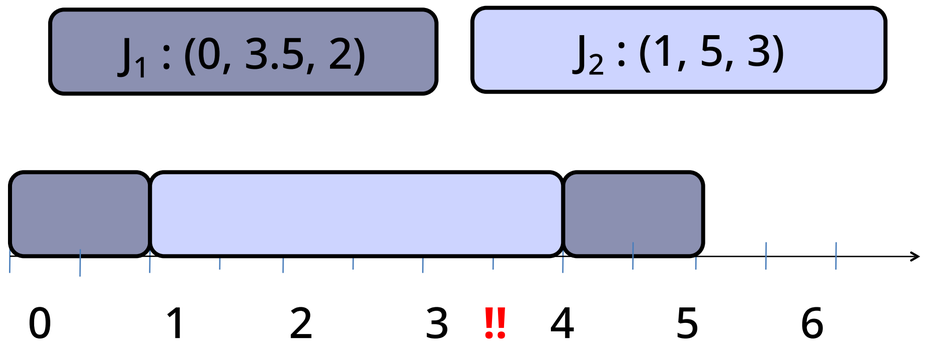
Least Slack Time First (LST)

scheduler checks slacks of all ready jobs and runs the job with the least slack

Two Versions:

- strict: slacks are computed at all (!) times
  - theoretical concept only, perfect processor sharing
  - real implementation may be:
    each instruction (prohibitively slow) or each timer tick
- non-strict: slacks are computed only at events (release, completion)

Example: Non-strict LST



Job: (release time, deadline, execution time)

t = 0: J_1 released and scheduled

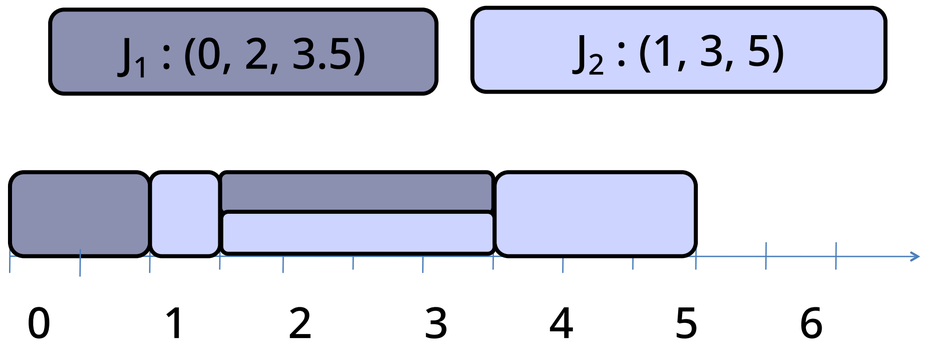
t = 1: J_2 released

Slack(J_1) = 3.5 – 1 – 1 = 1.5; Slack(J_2) = 5 – 3 – 1 = 1 → J_2 scheduled

t = 3.5: J_1 deadline miss

But EDF would schedule the jobs successfully!

Example: Strict LST



t = 0: J_1 released and scheduled

t = 1: J_2 released (see before) and scheduled

t = 1.5: Slack(J_1) = 3.5 – 1 – 1.5 = 1; Slack(J_2) = 5 – 2.5 – 1.5 = 1 →
J_1, J_2 are scheduled to execute in perfect interleaving (at half speed)

t = 3.5: J_1 completes → J_2 continued at full speed

t = 5: J_2 completes

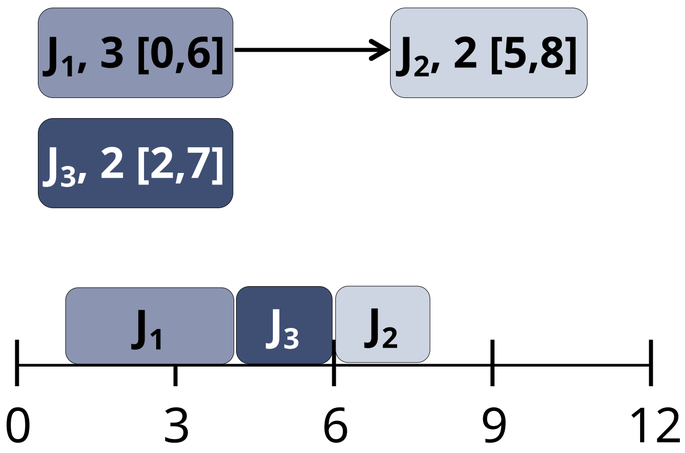
Latest Release Time (LRT)

Rationale

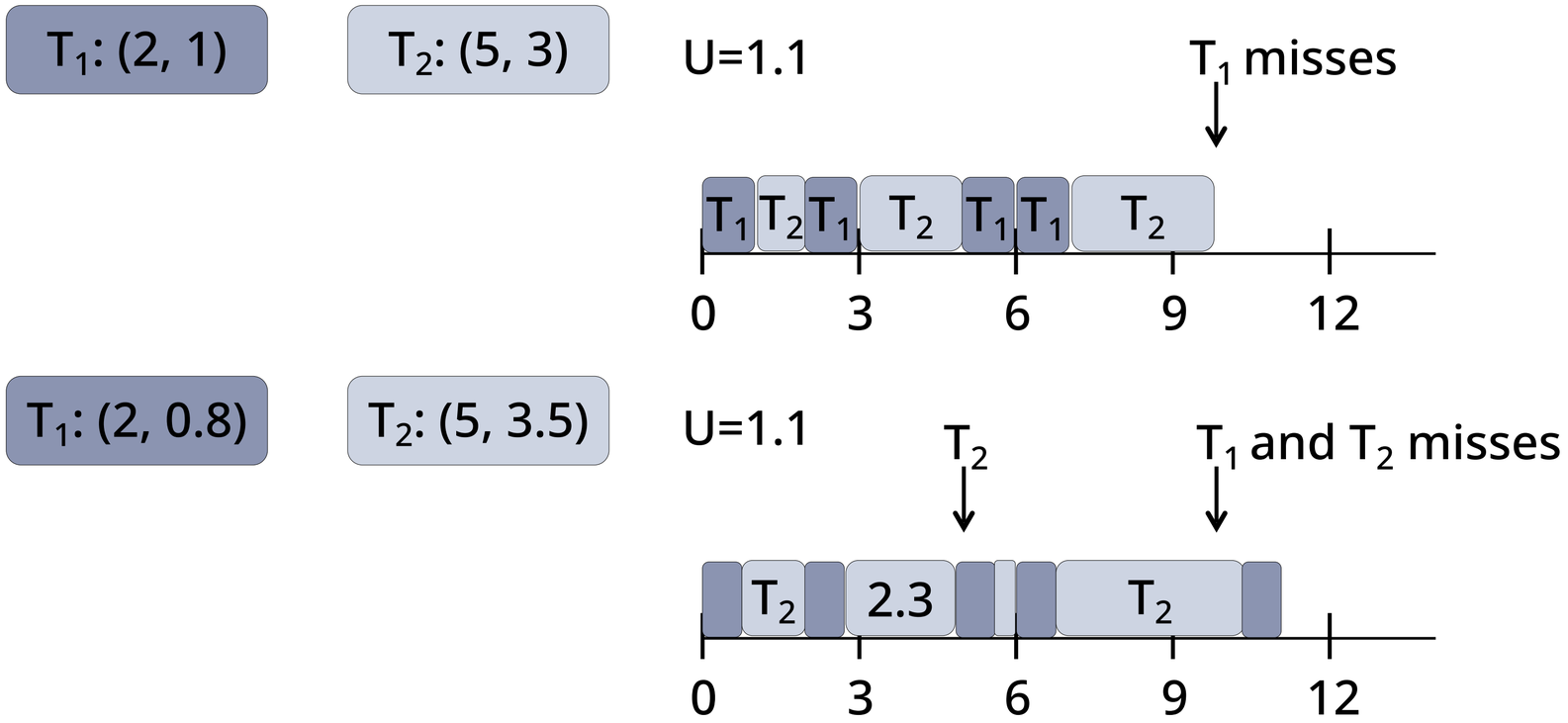
- no need to complete real-time jobs before deadline
- use time for other activities

Idea

- backwards scheduling:
  swap deadline/release, reverse precedence graph, use EDF
- run as late as possible
- use latest possible release times
- optimal



EDF and Overload

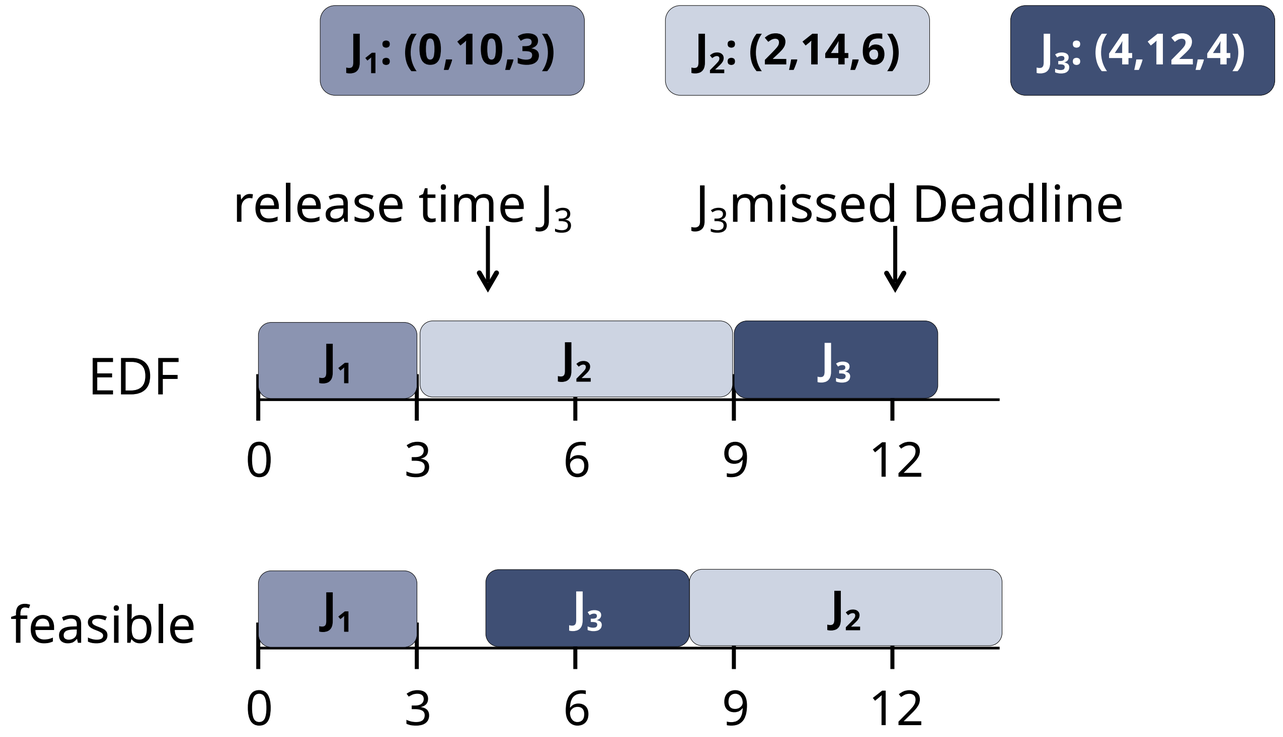


No easy way to determine which jobs miss deadline

In fixed priority systems: possible to predict the tasks affected by overruns

EDF and Non-Preemptivity

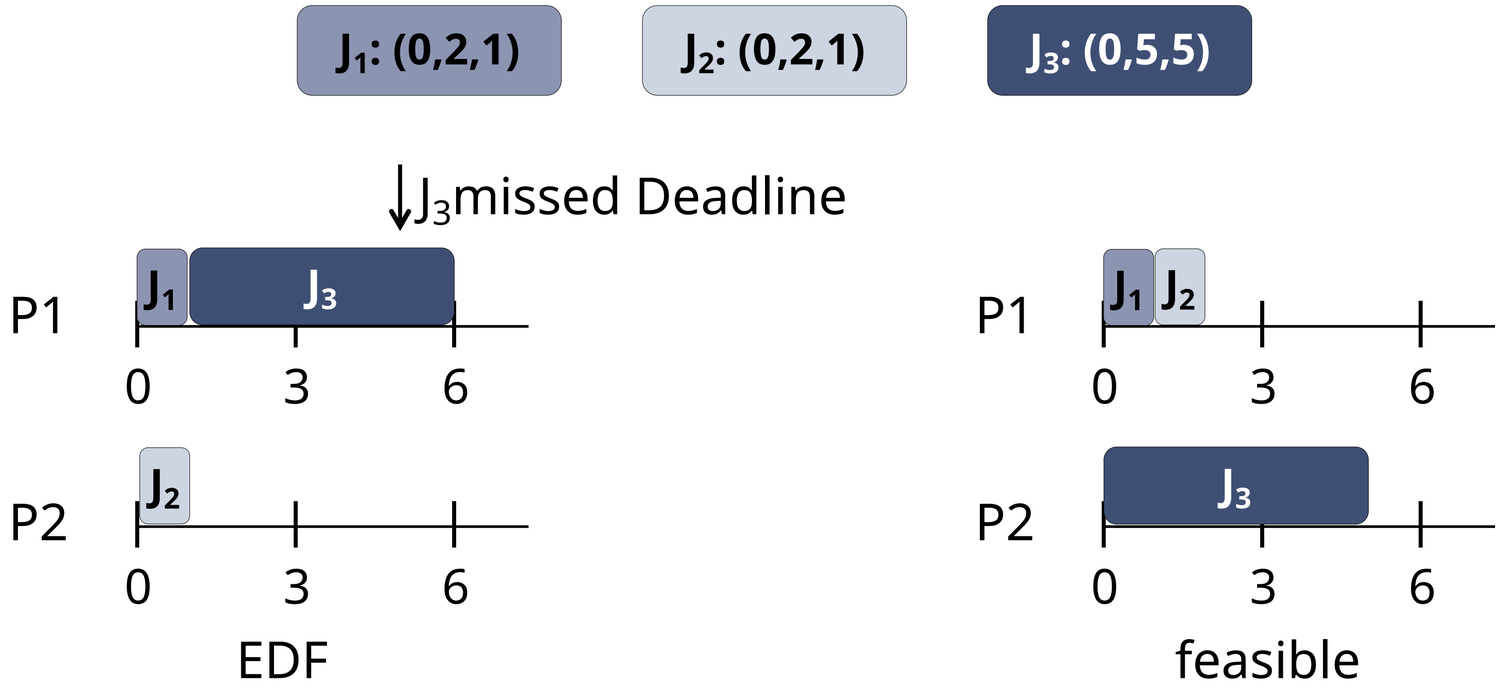
Job: (release time, deadline, execution time)



EDF is not optimal if jobs are not preemptable

EDF and Multiple Processors

Job: (release time, deadline, execution time)



- easy for time driven schedulers
- EDF is not optimal for multiprocessor systems

Summary



- schedulers: static and dynamic priorities
- classical algorithms: RMS, EDF
- schedulability analysis: utilization, critical instant
- RMS and EDF are optimal under simplistic assumptions