Real-Time Systems

Event-Driven Scheduling

Marcus Völp, Hermann Härtig

TECHNISCHE UNIVERSITÄT DRESDEN

WS 2013/14

Outline

- mostly following Jane Liu, Real-Time Systems
- Principles
- Scheduling
- EDF and LST as dynamic scheduling methods
- Fixed Priority schedulers
- Admission based on Utilization
- Few SMP insights (more later)
- Anomalies

Principles

Important properties:

- scheduling decisions are triggered by events (not time instants)
- events are release, completion, blocking, unblocking of jobs
- scheduler calls, interrupts, timers, … may trigger events
- scheduling decisions are on-line
  - scheduling must be simple
- admission is on-line or off-line
- *work-conserving* schedulers never leave a resource idle intentionally

Relaxing Restrictions of Time-Driven Systems

some restrictive assumptions of time-driven systems are relaxed:

- fixed inter-release times
  → minimum inter-release times
- fixed number of real-time tasks
  → no. of real-time and non real-time tasks can vary
- a priori fairly well known parameters
  → overload, schedule non-RT in the background, …

Principles

At Admission Time:

- assign jobs a value of a simple selection criteria: priorities
- check if feasible schedule exists for the selected scheduler

Scheduling / Dispatching:

- at event, select highest prioritized job

Principles

How good are schedulers?

- shorter response times
- more task sets
- higher utilization of resources

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).

Earliest Deadline First

Assign priorities at time when jobs are released:

“the earlier the deadline the higher the priority”

Theorem:

- one processor,
- jobs are preemptable,
- jobs do not contend for passive resources,
- jobs have arbitrary release times, deadlines,
- then: EDF is optimal
  (i.e. if there is a feasible schedule, there is also one with EDF)

[Least/Minimum] [Slack Time/Laxity] First

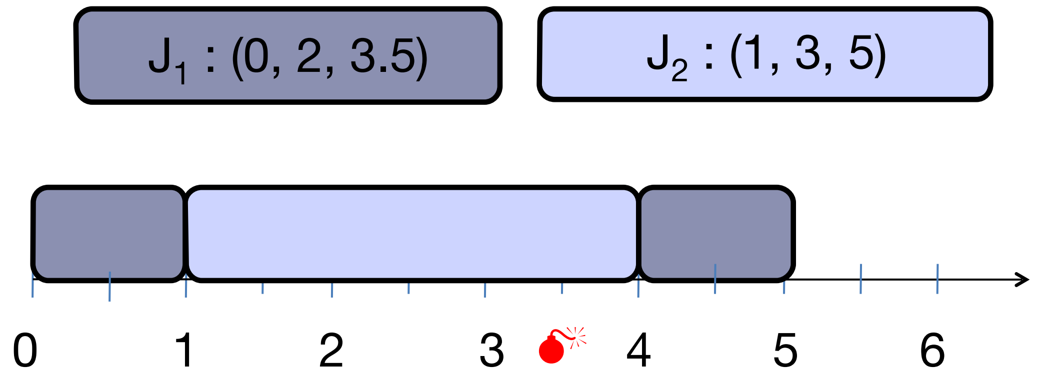
- Slack Time = Laxity:
  - (time to deadline -
    remaining execution time required to reach deadline)
- slack time: D - x - t
  - x remaining execution time of a job
  - D absolute deadline
  - t current time
- priority dynamic per job (see example)
- strict version is optimal

Least Slack Time First

- scheduler checks slacks of all ready jobs and runs the job with the least slack
- two versions:
  - Strict: slacks are computed at all times
    - Each instruction (prohibitively slow)
    - Each timer “tick”
  - Non-strict: slacks are computed only at events (release, completion)

Example: Non-strict LST

Job: (release time, execution time, deadline)



t = 0: J_1 released and scheduled

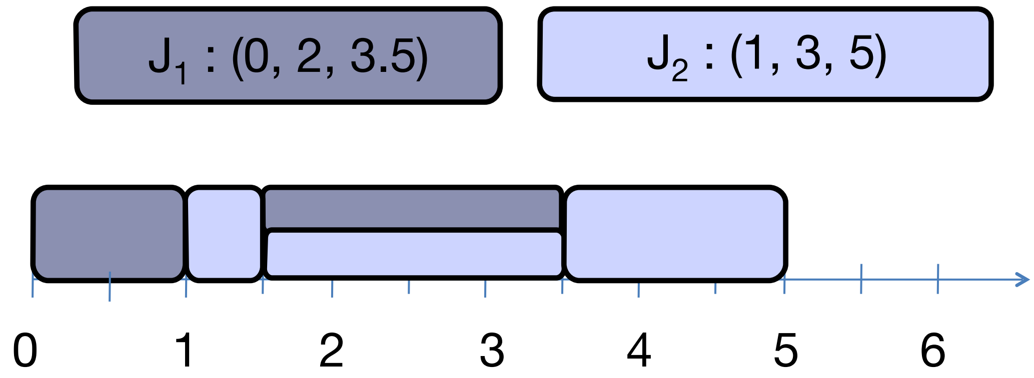
t = 1: J_2 released;
$L(J_1) = 3.5 - 1 - 1 = 1.5$; $L(J_2) = 5 - 3 - 1 = 1 \rightarrow J_2$ scheduled

t = 3.5: J_1 deadline miss

EDF schedules both jobs successfully!

Example: Strict LST

Job: (release time, execution time, deadline)



$t = 0$: J_1 released and scheduled

$t = 1$: J_2 released;
$L(J_1) = 3.5 - 1 - 1 = 1.5$; $L(J_2) = 5 - 3 - 1 = 1 \rightarrow J_2$ scheduled

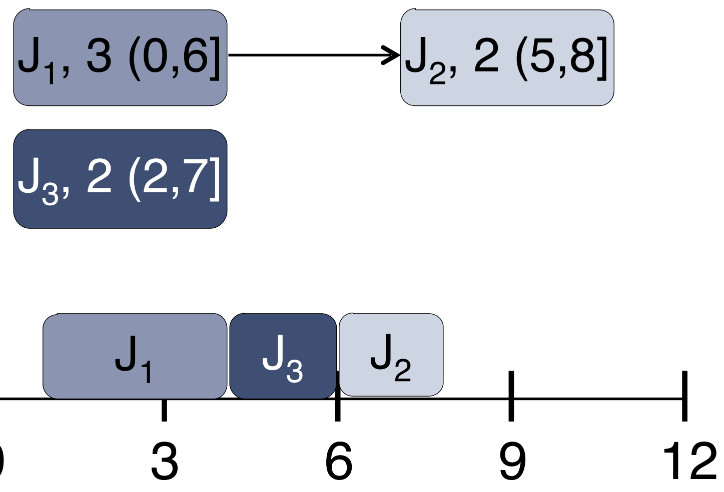
$t = 1.5$: $L(J_1) = 3.5 - 1 - 1.5 = 1$; $L(J_2) = 5 - 2.5 - 1.5 = 1 \rightarrow$
J_1, J_2 are scheduled and executed in parallel (at half speed)

$t = 3.5$: J_1 completes $\rightarrow J_2$ continued at full speed

$t = 5$: J_2 completes

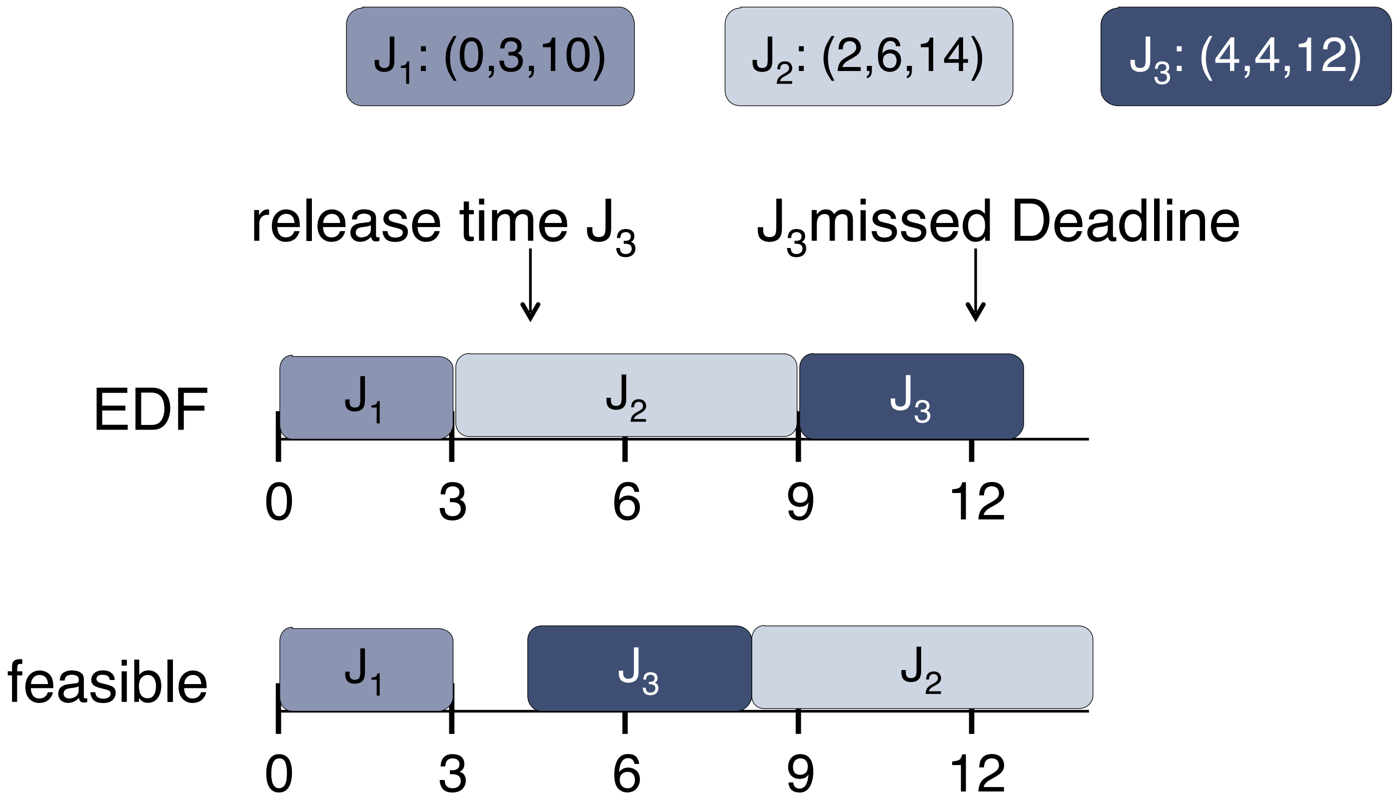
Latest Release Time (LRT)

- Rationale:
  - no need to complete real-time jobs before deadline
  - use time for other activities
- Idea:
  - backwards scheduling
    (Deadline <-> Release, turn around
    precedence graph, EDF)
  - run as late as possible
  - use latest possible release times
  - optimal (analog EDF and strict LST)



EDF and Non - Preemptivity

- Job: (release time, execution time, deadline)

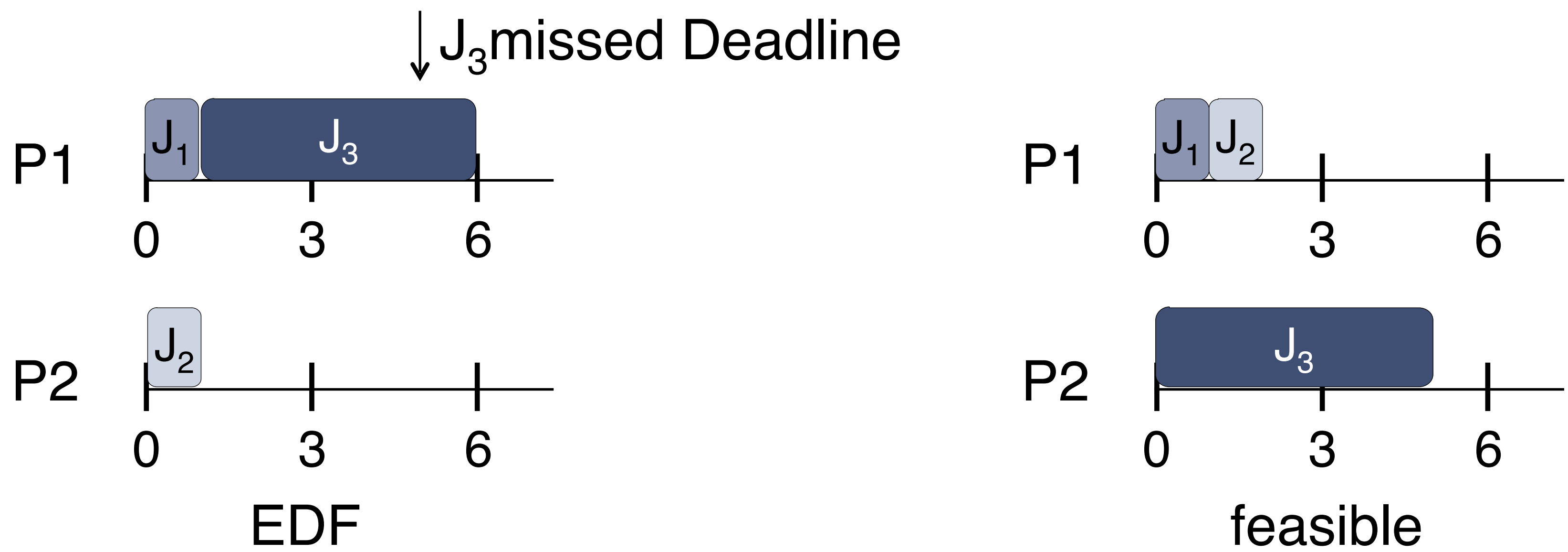


- EDF is not optimal if jobs are not preemptable

EDF and Multiple Processors

- Job: (release time, execution time, deadline)

J_1: (0,1,2) J_2: (0,1,2) J_3: (0,5,5)



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

Assumptions for Next Algorithms

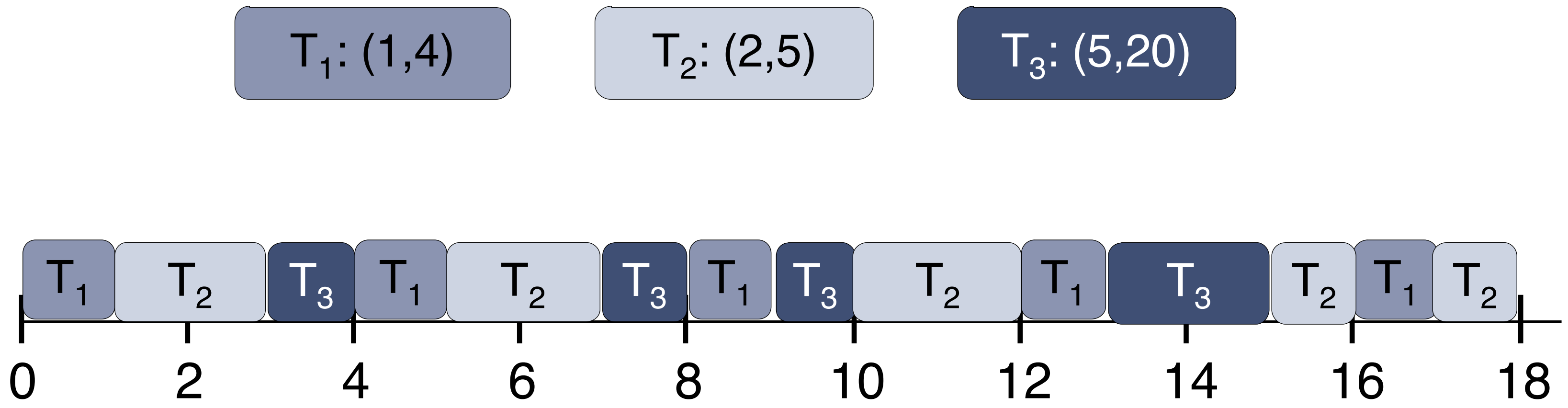
- Set of **periodic tasks** with these properties:
  - tasks are independent
  - one processor
  - no aperiodic tasks
  - preemptable, context switch overhead is negligibly small
  - period = minimum inter-release time
    (release times are not fixed but at least period apart)
- Since tasks are independent, tasks can be added (if admitted) and deleted at any time without causing deadline misses.

Priority-Driven Scheduling of Periodic Tasks

- To do:
  - priority assignment (off line / on line)
  - selection of next task (on line)
  - admission (required before new tasks are admitted)
- restrictions (whether they apply or not)
  - dependencies (precedence, sharing)
  - multiple processors
  - aperiodic, sporadic
- achievable resource utilization: $U = \sum_i \frac{e_i}{P_i}$

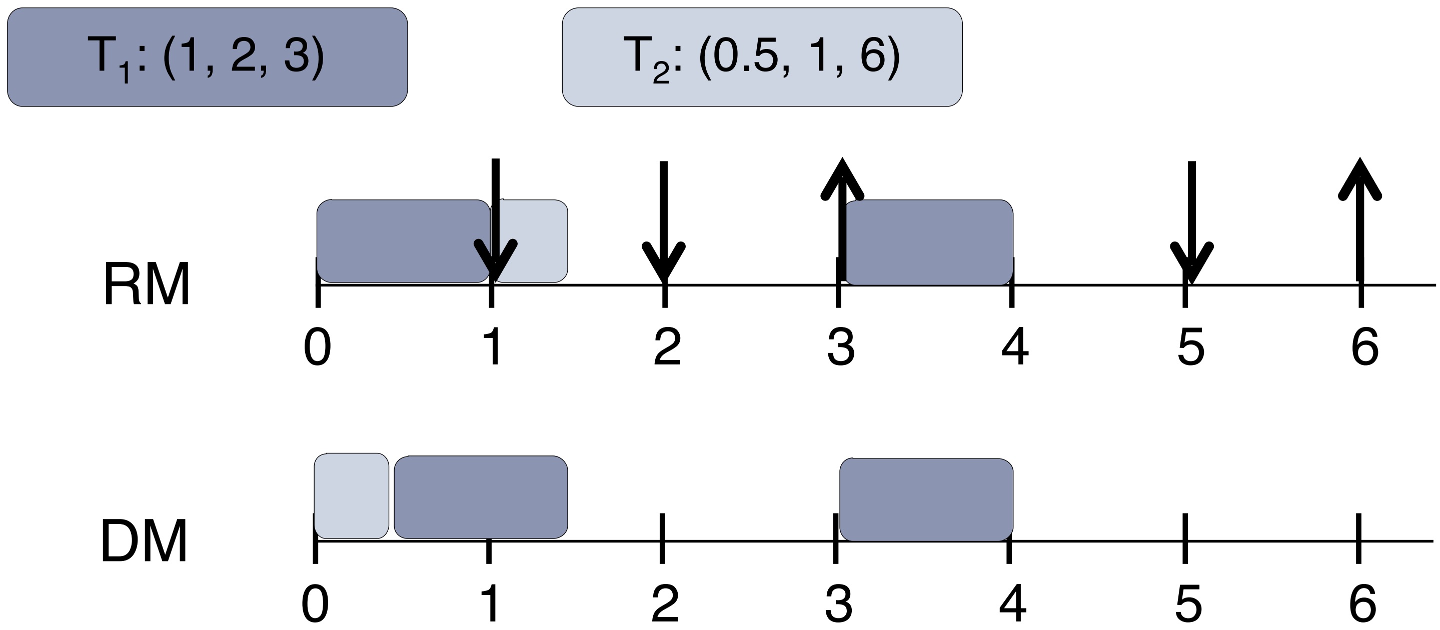
Rate Monotonic Scheduling

- fixed priority:
  - the shorter the period the higher the priority
    (rate: inverse of period)



Deadline Monotonic Scheduling

- fixed priority:
  - the shorter the relative deadline the higher the priority
- example: (e, D, P)



- Conclusion (no proof):
  RM not optimal but DM if D ≤ P for all tasks

Optimality of Fixed Priority Schedulers

T: periodic tasks, independent, preemptable, one CPU

Deadline Monotonic:

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

Rate Monotonic (RMS):

- relative deadlines = periods
  if there is any feasible fixed priority schedule for T,
  then Rate Monotonic produces a feasible as well

Admission based on Utilization

- A task (P,e) requires e/P of the capacity of a processor.
- Any scheduler can admit at most up to full capacity:
  - For a task set $T_1 \dots T_n$: $\sum e_i/P_i \le m$ is a necessary but not sufficient condition for m processors.
- Can we establish a maximum bound X such that

  $T_1 \dots T_n$: $\sum e_i/P_i \le X$

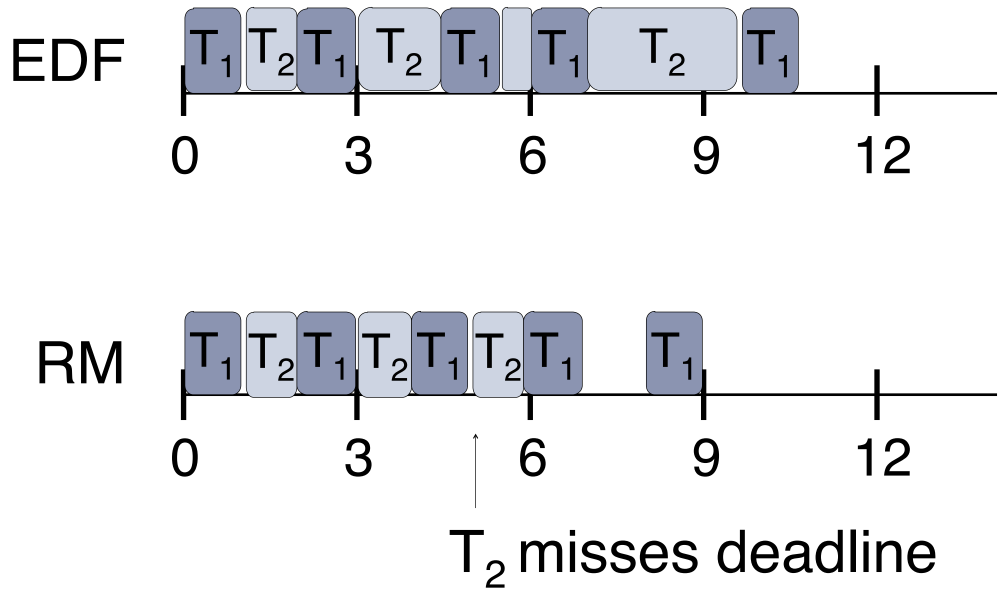
  is sufficient?

Such bounds are called *schedulable utilization* SU.

- SU depends on the scheduling algorithm.
- the higher the better.

Utilization: RMS / EDF

T_1: (2 , 1) T_2: (5 , 2.5) U=1



RMS not optimal in general

Some Schedulable Utilization (SU) Results

- independent tasks, preemptable,
  relative deadline = period, m = 1 processor
- n ... Number of Tasks
- EDF: SU = 1
- RMS: $SU = n\,(2^{1/n} - 1)$ $n \rightarrow \infty : \ln(2)$
- RMS with harmonic periods: SU = 1
- harmonic periods (also called simply periodic):
  for all pairs of tasks Ti,Tj: if Pi <= Pj then Pj = nij* Pi

(Fixed Priority) Schedulability and Blocking

- Ti may have to wait for non-preemptable, lower priority task
- b_i: longest non-preemptable portion of all lower priority jobs
- schedulability $SU_x(i)$ for all tasks T_i with fixed priority scheduler x:
  - schedulable utilization for scheduling method x with i tasks:
  - $U_i = e_1/P_1 + e_2/P_2 + \ldots + e_i/P_i$
  - $U_i + b_i/P_i \leq SU_x(i)$

Non Negligible Context Switch Time

- For Job level fixed priority schedulers:
    - i.e. each job preempts at most one other job
- 2 context switches:
    - release (when it preempts other)
    - completion

- include context switch overhead in WCET:
    - $WCET_i := WCET_{i\_original}$ + 2 context switches

Static and Dynamic (priority)

If no new tasks arrive: static vs. dynamic priorities

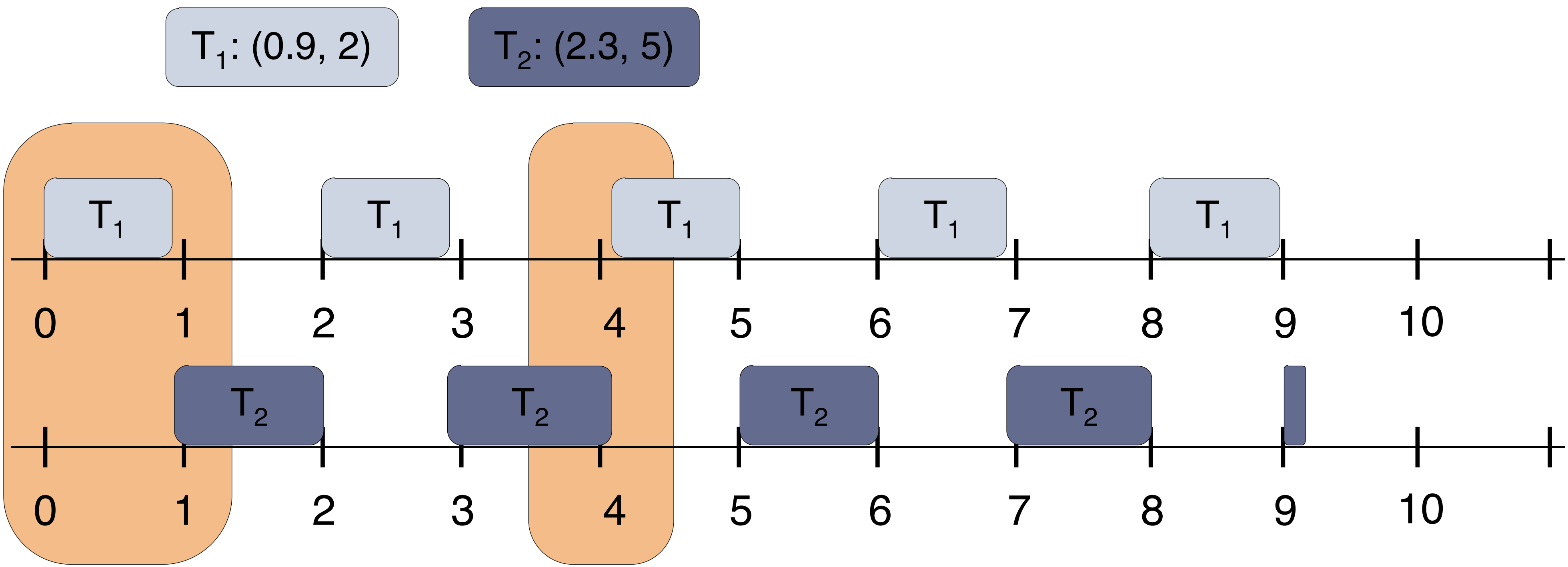
- Task static: Task T does not change its priority, i.e. all jobs of T have same fixed priority
- Job static: Jobs do not change their priorities
- Job dynamic: Jobs change their priorities

Careful:

Job static is often called dynamic as well

Earliest Deadline First, priority assignment:

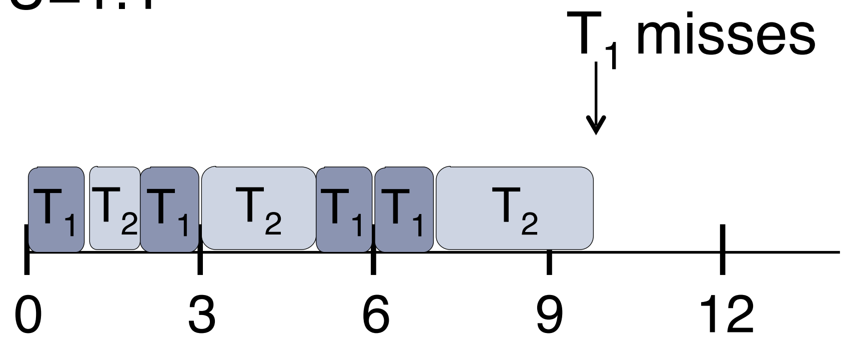
- fixed per job, dynamic at task level:
  - the nearer the absolute deadline of a job at release time
    the higher the priority



EDF and Overload, examples

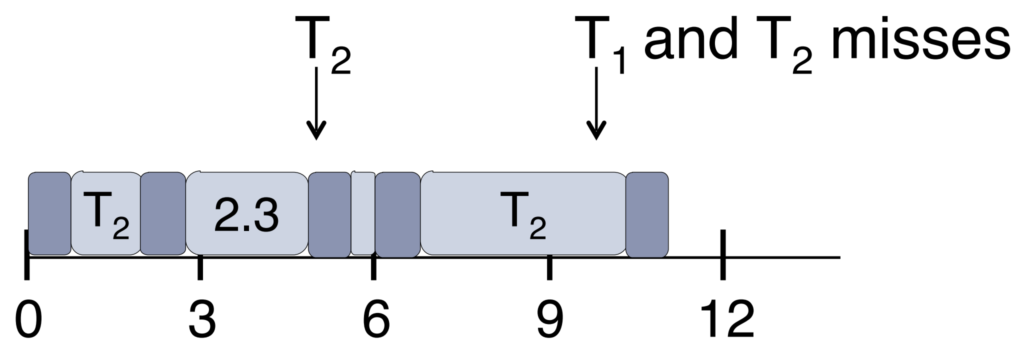
T_1: (1, 2) T_2: (3, 5)

U=1.1



T_1: (0.8, 2) T_2: (3.5, 5)

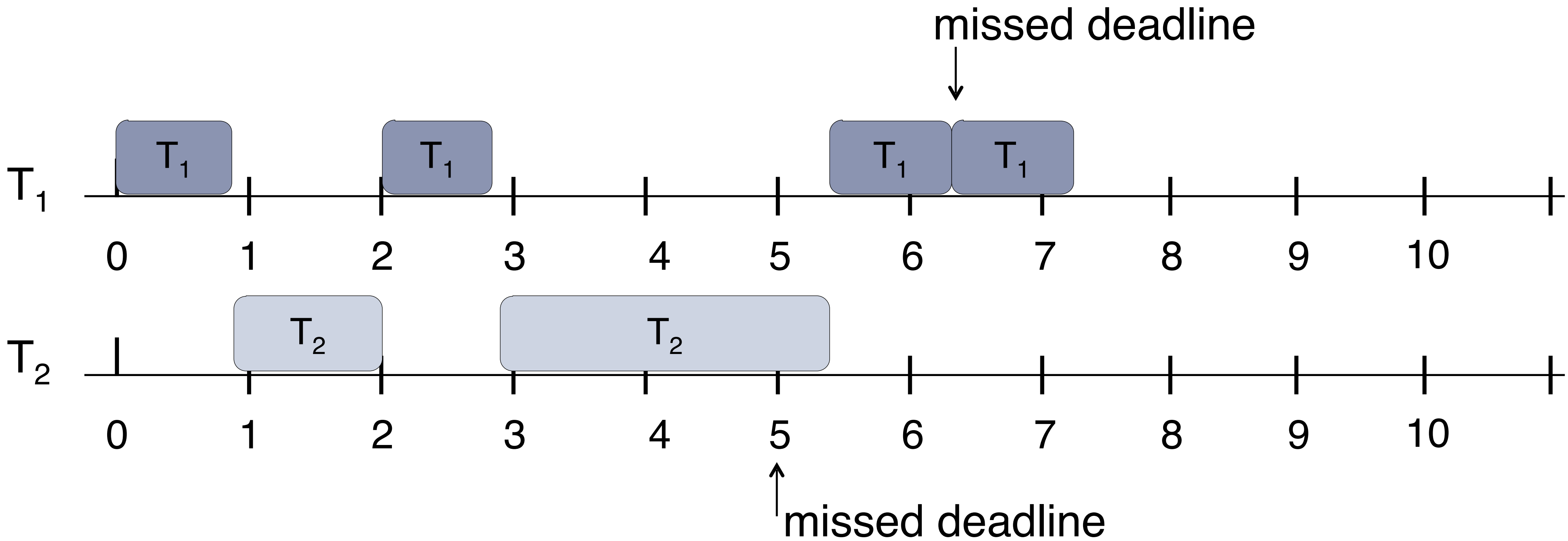
U=1.1



No easy way to determine which jobs miss deadline

EDF and Overload, one more example

T_1: (0.8, 2) T_2: (4.0, 5)   U=1.2



in fixed priority systems it is possible to predict which tasks are affected by overruns

(Fixed Prio and) Limited Priority Levels

- Required: Mapping of
  - Scheduling-Priorities: 1 ... n to
  - Operating System Priorities: Π_1, Π_2, … Π_m
- Jobs running with same OS-Prio but different Sched-Prio use:
  - FIFO, Round Robin, …
- Schedulability loss
  - Notation: Π_i as grid on Scheduling Priorities
  - Example: 10 scheduling priorities, 3 OS priorities
    - possible mapping: $\Pi_1=3$, $\Pi_2 = 8$, $\Pi_3 = 10$
    - Interpretation: 1,2,3 mapped to Π_1, 4,5,6,7,8 to Π_2, 9,10 to Π_3
- How is the Schedulability Test affected?

(Fixed Prio and) Limited Priority Levels

- Mappings:
  - uniformly distributed: $k = \left\lfloor \frac{n}{m} \right\rfloor$

    Scheduling Priority Π mapped to k $\left\lfloor \frac{\Pi}{m} \right\rfloor$
  - constantratio:
    keep $(\Pi_{i-1} + 1) / \Pi_i$ as equal as possible

Schedulalibility Loss

- Rate Monotonic, large n ...
  - $g = \min((\Pi_{i-1} + 1) / \Pi_i)$
  - $SU_{RM} = \ln(2g)+1-g$
- relative schedulability (rs): relation to ln(2)
- Example:
  - n = 100000, m = 256
  - rs = 0.9986
- only small schedulability loss with 256 priorities

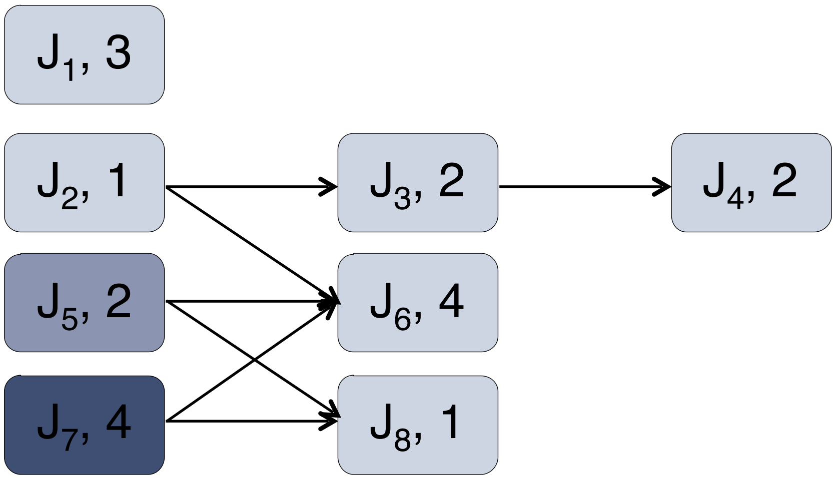
Predictable/Sustainable Execution

Informal definition:

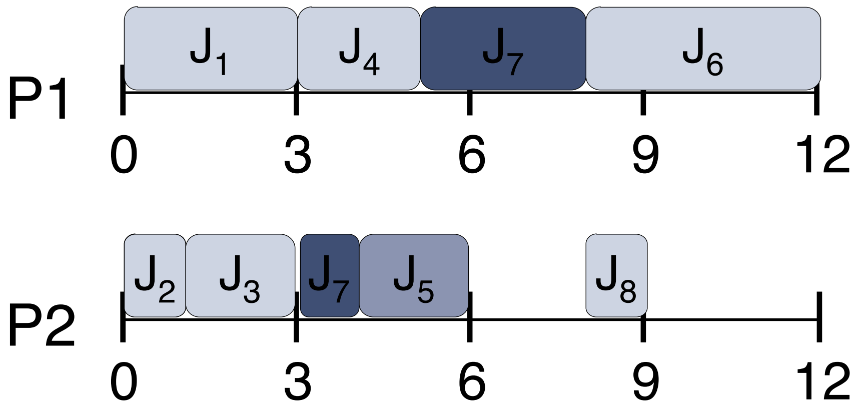
- Given a set of periodic tasks with *known minimal and maximal* execution times and a scheduling algorithm.
- A schedule produced by the scheduler when the execution time of each job has its maximum (minimum) value is called a *maximum (minimum) schedule.*
- An execution is called *predictable,* if for each actual schedule the start and completion times for each job are bound by these times in the *minimum and maximal 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 one processor.

Preemptive vs. Non-Preemptive Scheduling

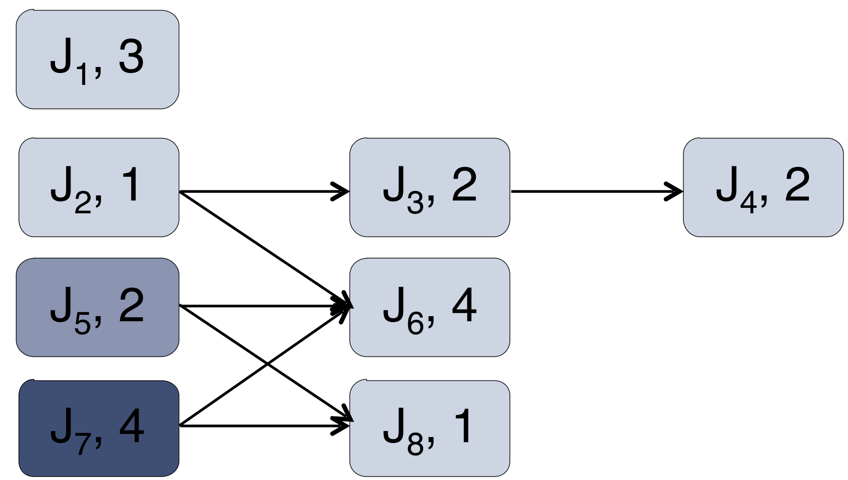
- 2 processors,
- Tasks: notation used below: J_i ,e_i
  - release time of J_5 is 4, all others 0; (!)
- static priorities, assigned such that:
  i < k => Prio(J_i) higher than Prio(J_k)
- Jobs can "migrate"
- precedence graph:

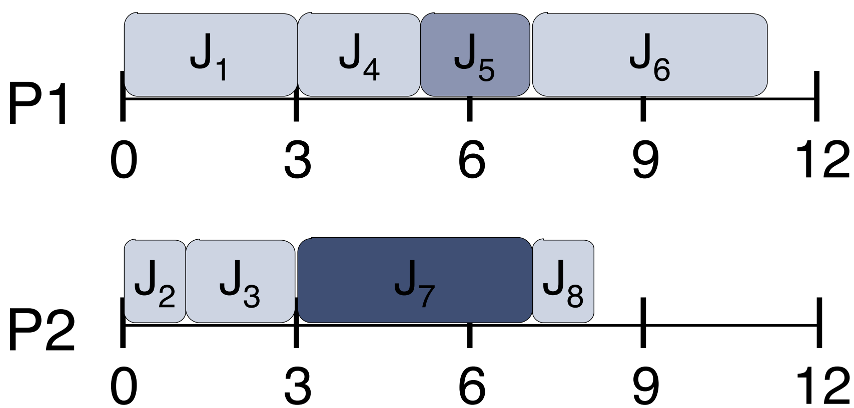


Example, executions



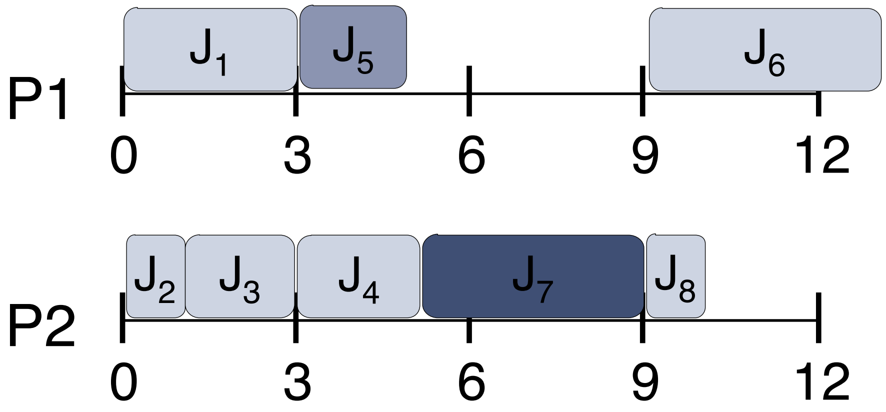
preemptiv



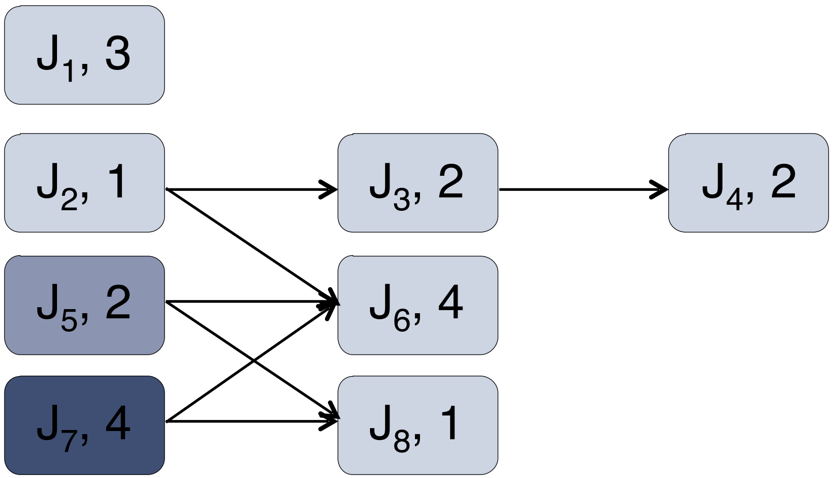


non-preemptiv

Modified Example: release time of J_5 = 0



non-preemptiv



Which is better?

- No general answer known!
- If jobs have same release time:
  preemptive is better (or equal) in a multiprocessor system if cost for preemption is ignored
- more precise: *makespan* is better
  (makespan = response time of job that completes last)
- how much better? Coffman and Garey:
  2 processors:
  makespan(non-preemptive) <= 4/3 * makespan(preemptive)

Multiple Processors

- Static vs dynamic allocation to processors
  - Partitioned   tasks are assigned to processors
  - Static: jobs are assigned to processors once
  - Dynamic: jobs “migrate”
    - example: one run queue served by all processors
- EDF not optimal
  general: “static-job” scheduling not optimal
- There are optimal “dynamic-job” schedulers

Lessons Learned

- Schedulers: static, static and dynamic (RMS, EDF, LST)
- Schedulability Analysis: Schedulability Utilization
- RMS and EDF are optimal under simplistic assumptions
- Anomalies