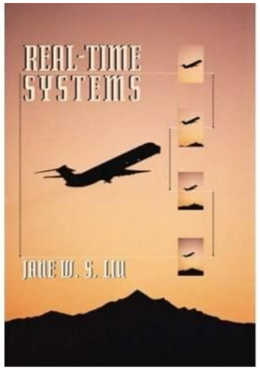
Real-Time Systems

Part 7: Scheduling

Content

Literature



- Jane W. S. Liu, Real-Time Systems, 2000
- Fridolin Hofmann: Betriebssysteme - Grundkonzepte und Modellvorstellungen, 1991
- Klaus Gresser, Echtzeitnachweis ereignisgesteuerter Realzeitsysteme, Dissertation, TUM, 1993

Journals:

- *Baker, T.P.; Shaw, A.; , "The cyclic executive model and Ada," Real-Time Systems Symposium, 1988., Proceedings. , vol., no., pp.120-129, 6-8 Dec 1988*
- John A. Stankovic, Marco Spuri, Marco Di Natale, and Giorgio C. Buttazzo: Implications of classical scheduling results for real-time systems. IEEE Computer, Special Issue on Scheduling and Real-Time Systems, 28(6):16–25, June 2005.
- Giorgio C. Buttazzo: Rate Monotonic vs. EDF: Judgement Day (http://www.cas.mcmaster.ca/~downd/rtsj05-rmedf.pdf)
- Puschner, Peter; Burns, Alan: A review of Worst-Case Execution-Time Analysis, Journal of Real-Time Systems 18 (2000), S.115-128
- *Liu, C. L.; Layland, J. (1973), "Scheduling algorithms for multiprogramming in a hard real-time environment", Journal of the ACM* ***20*** *(1): 46–61*
- *Lehoczky, J.; Sha, L.; Ding, Y.; , "The rate monotonic scheduling algorithm: exact characterization and average case behavior," Real Time Systems Symposium, 1989., Proceedings. , vol., no., pp.166-171, 5-7 Dec 1989*

Introduction

Scheduler and Dispatcher

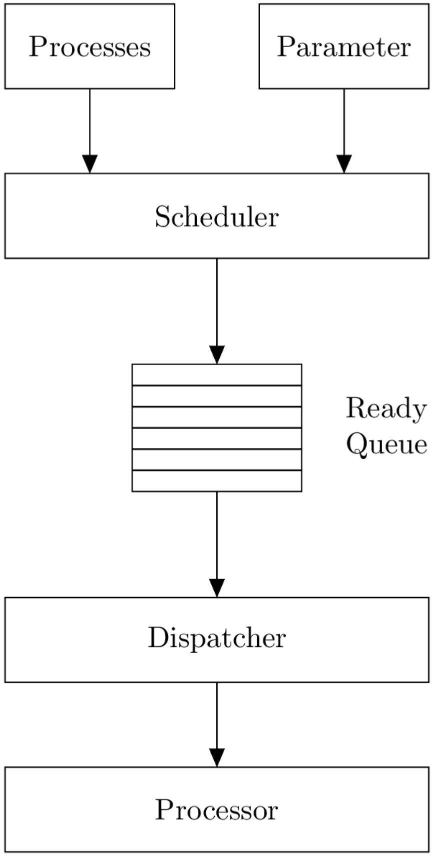
- **Scheduler:**

  If a resource is to be used by many consumers, access to the resource has to be coordinated. This resource *allocation* is performed by a ***scheduler***.

  In computer systems, the term scheduler often refers to the CPU scheduler which controls the allocation of the CPU to ***tasks***.

- **Dispatcher:**

  While the scheduler plans the CPU allocation, the dispatcher executes the scheduler plan by:
  - Switching the context
  - Switching to user mode
  - Jumping to the proper location in the user program to restart it

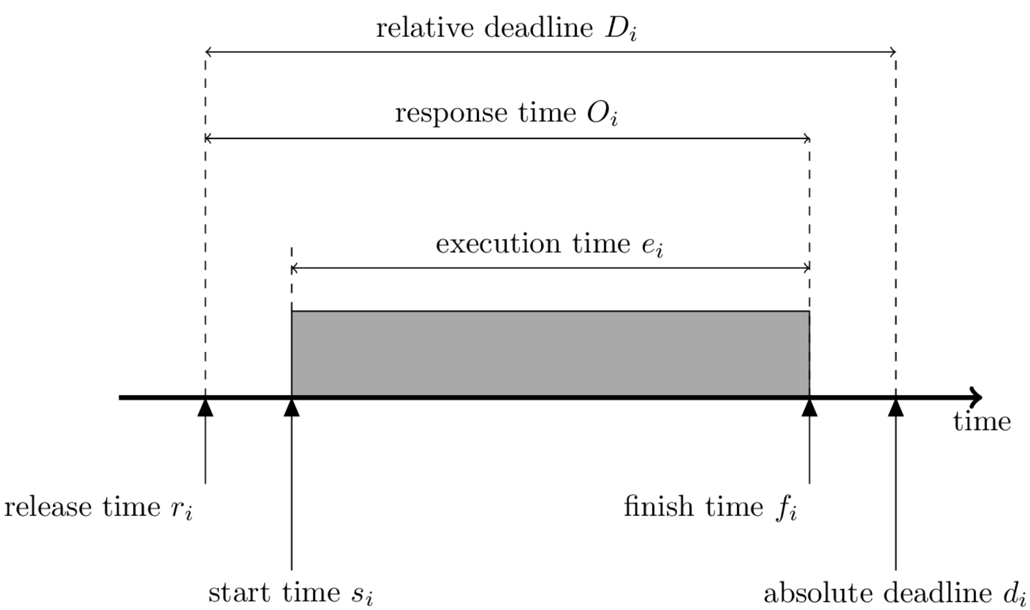


Introduction

Task Model

We introduce the following model for a task:

- **Release Time (*or* arrival time) r_i**
  Earliest time at which task *i* is enabled.
- **Start Time s_i**
  Time at which execution of task starts.
- **Finish Time f_i**
  Time at which task completes execution.
- **Response Time O_i**
  Interval between release and finish time.

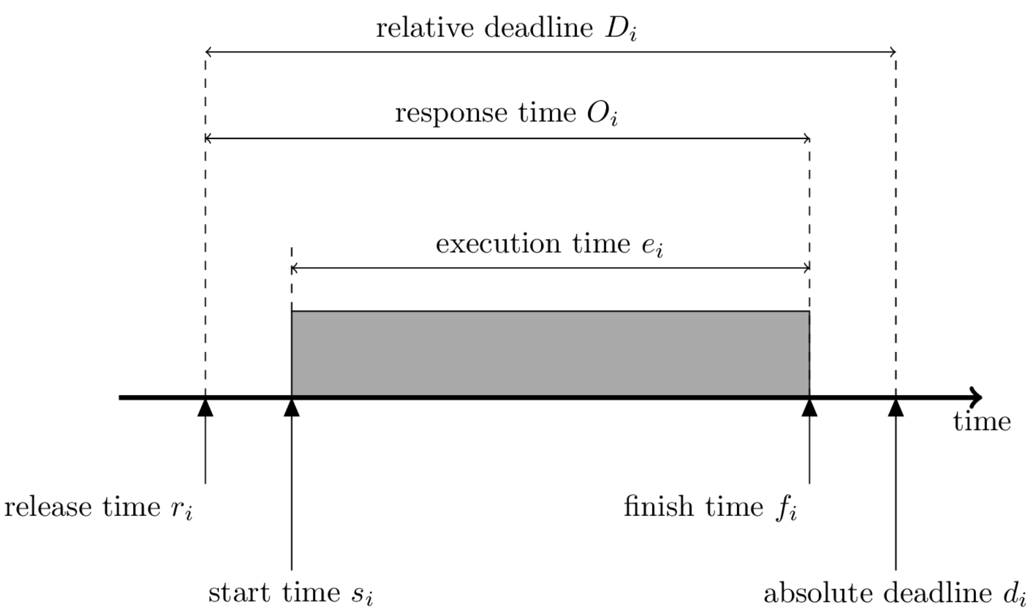


Introduction

Task Model (continued)

We introduce the following model for a task:

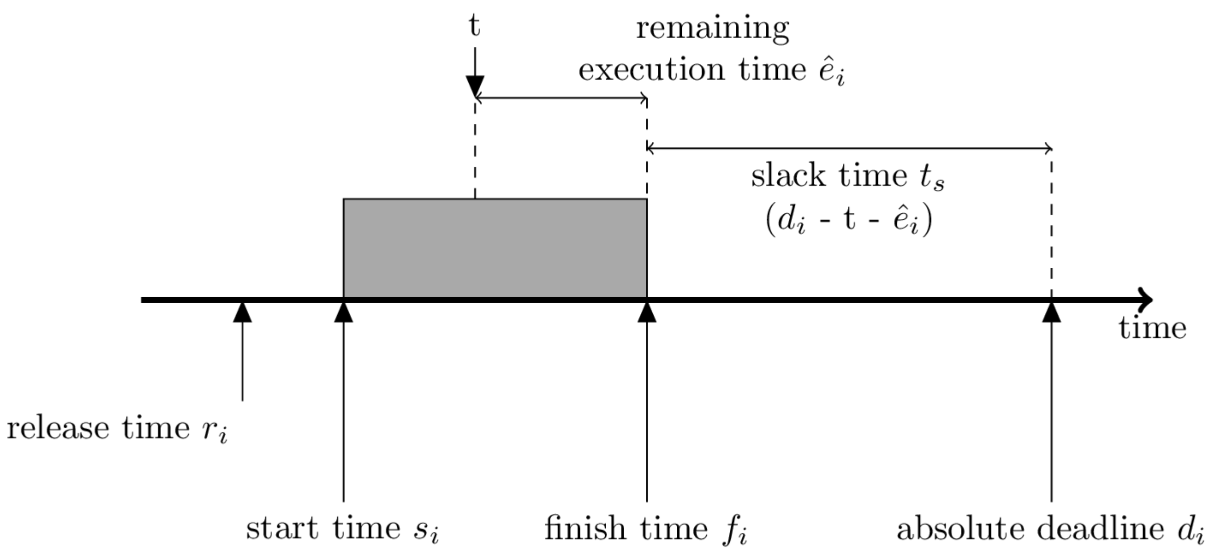
- **Execution Time e_i**
  (remaining execution time $\hat{e}_i$ – see next slide)
  Total time of task execution (does not include durations where the task was blocked).
- **Relative Deadline D_i**
  (absolute deadline d_i)
  The relative deadline is the maximum tolerated response time.
- **Tardiness**
  Measures the deadline violation.
  0 if $f_i \leq d_i$, otherwise $f_i - d_i$



Introduction

Task Model (continued)

- Slack time t_s



Introduction

Task Model (continued)

- Preemptable Task
  A task is called ***preemptable*** if its execution can be suspended.
  - ***Fully preemptable***: preemption can occur at any time
  - ***Preemption Points***: preemption can only occur at predefined times

- Periodic Task
  A task is called **periodic**, if it is released with a fixed frequency (or period p).

- Aperiodic Task
  A task is called ***aperiodic***, if it either has a soft deadline or no deadline at all.

- Sporadic Task
  A task is called ***sporadic***, if it has a hard deadline but is released at random times.

Introduction

Feasible, Optimal Schedule & Schedulability Test

- **Feasible Schedule**

  A schedule is called ***feasible***, if all tasks of the task set $T_i, i \in \{1,2,\ldots,k\}$ that share the CPU meet their deadlines:

  $$O_i \leq D_i, \forall i \in \{1,2,\ldots,k\}$$

- **Optimal Scheduler**

  We call a scheduler ***optimal*** if the algorithm always produces a feasible schedule given that a feasible schedule exists for the task set.

- **Schedulability Test**

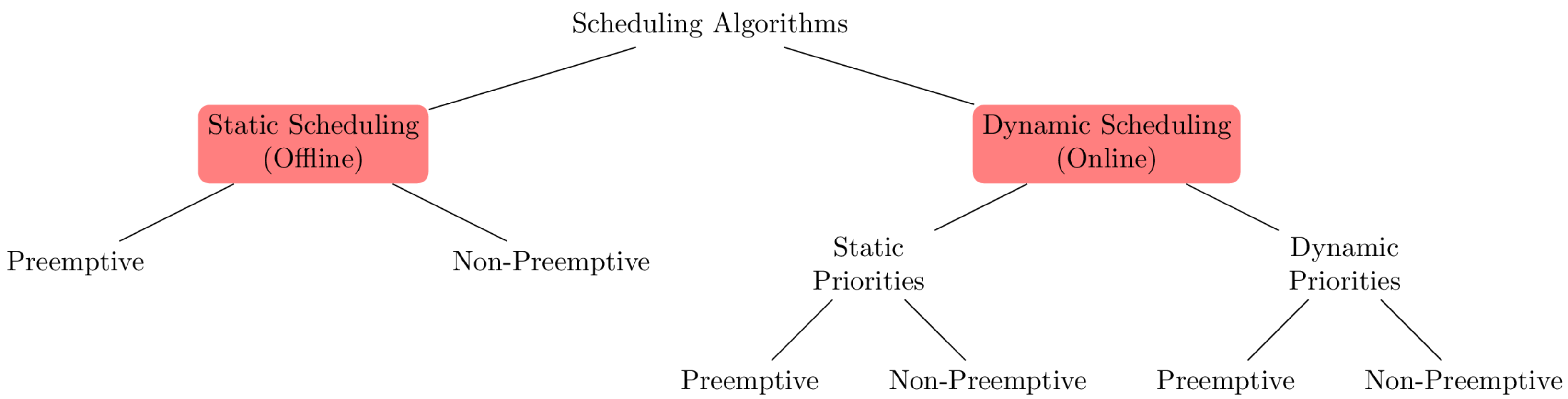
  A schedulability test varifies whether a feasible schedule exists for a particular task set.

Content

Scheduling Algorithms

Overview

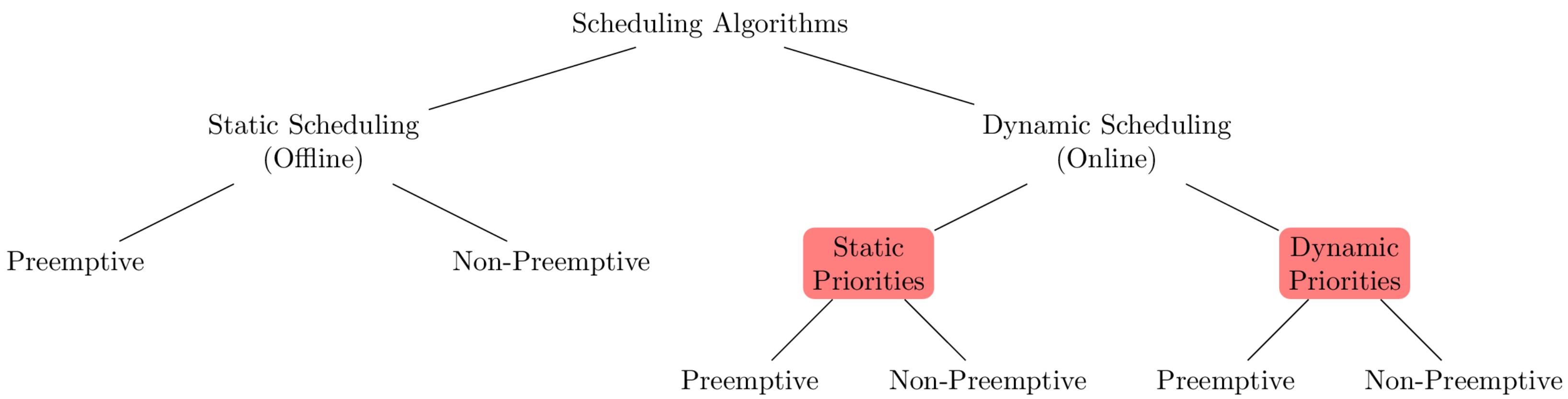
- **Static Scheduling (Offline)**
  A static scheduling is defined at compile time (offline). All tasks as well as important parameters (e.g. execution times) need to be known a priori.
- **Dynamic Scheduling (Online)**
  A dynamic scheduling is performed at runtime, based on the current set of active tasks and their resource dependencies.



Scheduling Algorithms

Overview

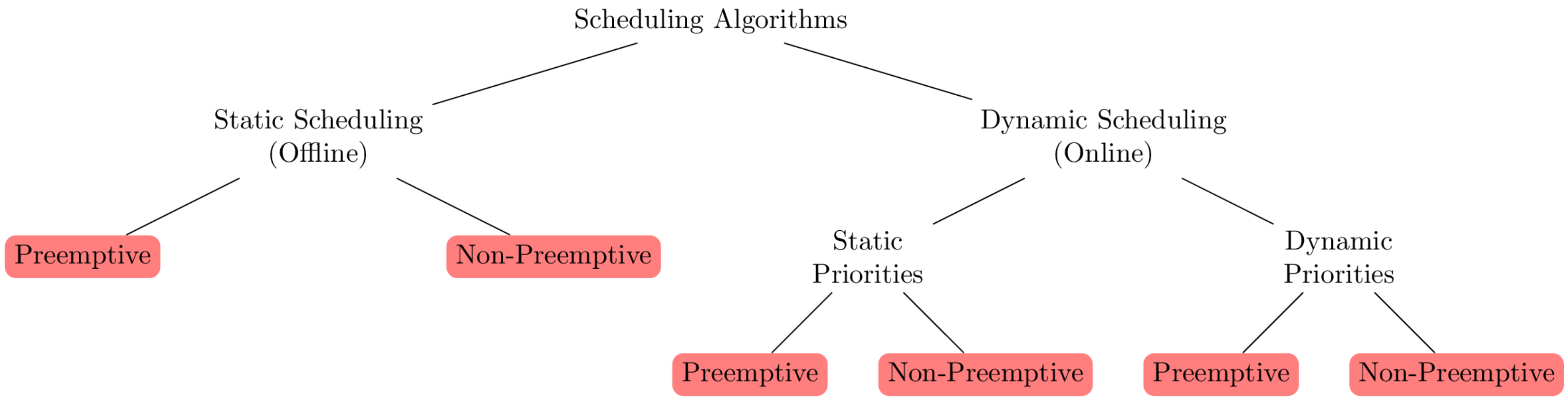
- **Static Priorities**
  Priority of task depends on task parameters that are known a priori (e.g. deadline or period) and does not change over runtime.
- **Dynamic Priorities**
  Priority of task changes at runtime depending on dynamic parameters (e.g. currently allocated resources).



Scheduling Algorithms

Overview

- **Preemptive**
  A scheduler is called preemptive, if it is able to interrupt the execution of a task and to re-assign the CPU.
- **Non-Preemptive**
  A scheduler is called non-preemptive if it executes a once started task until it finishes or blocks.



Content

Clock-Driven Scheduling

Notations and Assumptions

- The clock-driven scheduling approach is only applicable if the system is deterministic.
- **Assumptions:**
  - There are *n* periodic tasks in the system.
  - The parameters of all tasks are known a priori.
- **Periodic task model notation:**
  - There are *n* periodic tasks T_i, defined by the 4-tuple:

    $$T_i: (\phi_i; p_i; e_i; D_i)$$

    where ϕ_i is the phase and p_i is the period of the periodic task.
  - If the phase is 0, we will omit it.
  - If the period is equal to the relative deadline, we will omit D_i.

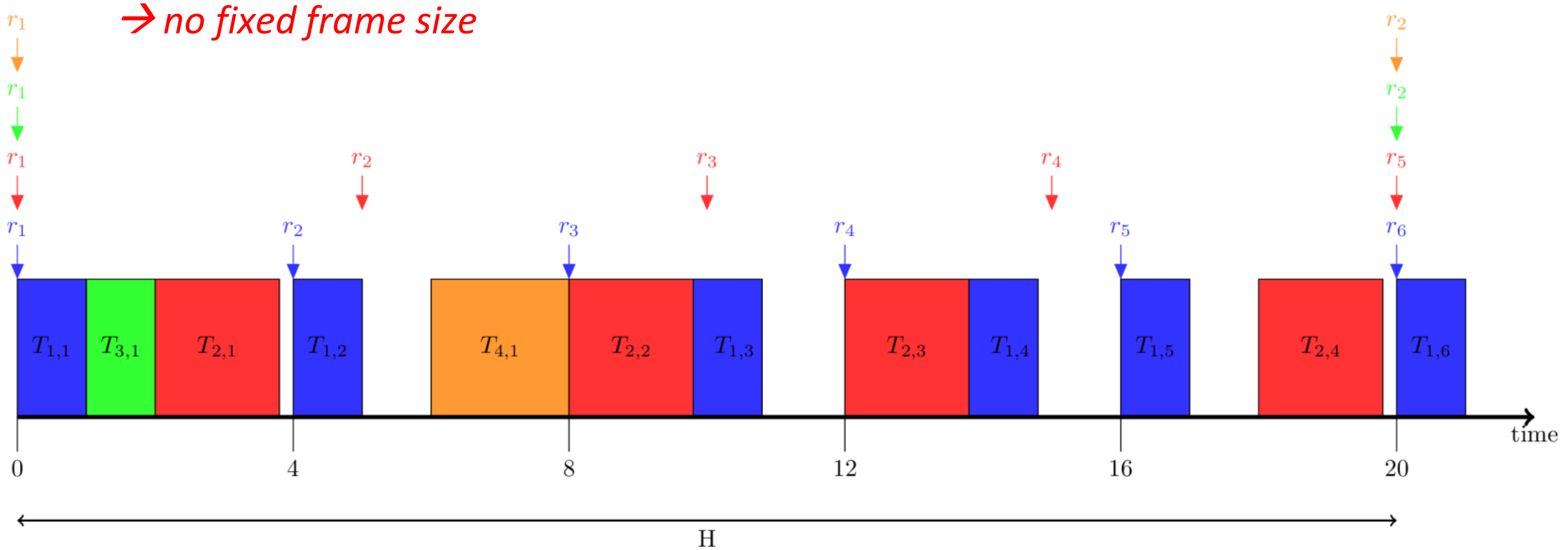
Clock-Driven Scheduling
Variable Frame Length Schedule

- A ***frame*** is the time interval after which the scheduler will be triggered.
- The length of a frame is called the **frame size *f*.**
- *Example of a static scheduler with a* ***variable*** *frame size f:*
  - *Given are four independent periodic tasks that are executed on a single-processor system:* $\boldsymbol{T_i=(p_i, e_i)}$
    - *T1 = (4, 1)*
    - *T2 = (5, 1.8)*
    - *T3 = (20, 1)*
    - *T4 = (20, 2)*

Clock-Driven Scheduling

Variable Frame Length Schedule

- *Example (continued):*
  - *The hyperperiod H (the least common multiple of all p_i) is 20*
  - *A possible static schedule is shown in the following figure (if no task is running the **Idle-Task** is executed):*
  - *The scheduler is called at times: 0, 1, 2, 3.8, 4, 6, etc.*
    → no fixed frame size



Clock-Driven Scheduling

Fixed Frame Length Schedule

- Ideally, we want to ensure that the cyclic schedule has some desired characteristics, e.g. a constant frame size.
- An optimal, constant frame size can be computed from a task set T_i by taking the following constraints into account (Baker and Shaw, 1988):
  - Constraint 1: The frame size should be smaller than or equal to the relative deadline D_i:

$$f \le \min_{1 \le i \le k}(D_i)$$

  - Constraint 2: Ideally, the frame size should be large enough to execute the longest task within one single frame:

$$f \ge \max_{1 \le i \le k}(e_i)$$

Clock-Driven Scheduling

Fixed Frame Length Schedule

– Constraint 3: The hyperperiod *H* should be an integer multiple of the frame size *f*:

$$F = \frac{H}{f} \text{ with } F \in N$$

(The relevant frame sizes f can easily be determined by computing all integer factors of the periods of the tasks)

– Constraint 4: The frame size *f* has to be small enough to ensure that no task misses its deadline (between the release time and the deadline has to fit at least one frame):

$$2f - GCD(p_i, f) \leq D_i$$

(GCD = Greatest Common Divisor)

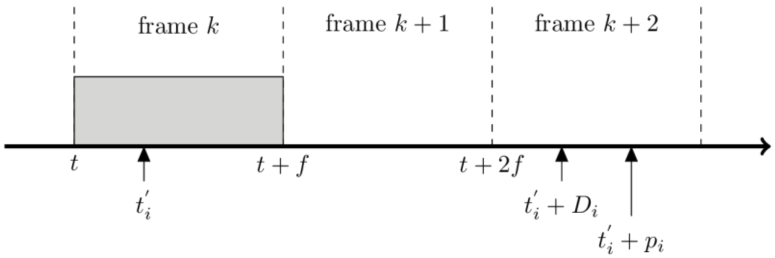
Clock-Driven Scheduling

Fixed Frame Length Schedule

– Constraint 4 – Explanation

$$t + 2f \leq t_i^{'} + D_i$$

$$2f - (t_i^{'} - t) \leq D_i$$

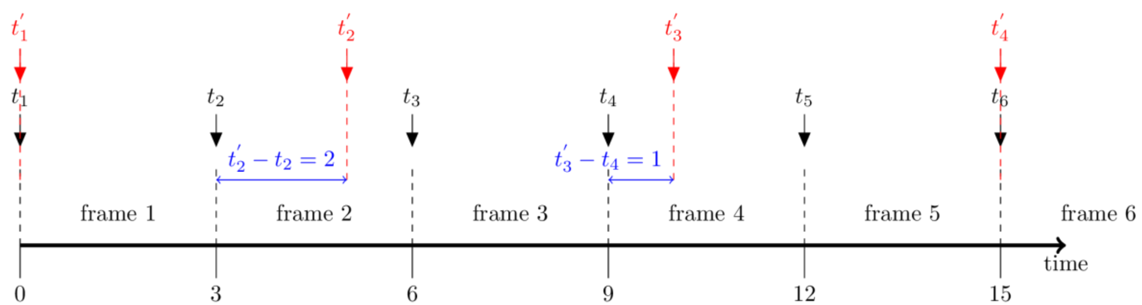


As we are interested in the upper limit of *f*, we have to compute the smallest possible value of *(t'$_i$-t)* larger than 0: This is the greatest common divisor of p_i and *f*: $2f - GCD(p_i, f) \leq D_i$

Example:

T with period 5

Frame size f = 3



Clock-Driven Scheduling

Fixed Frame Length Schedule

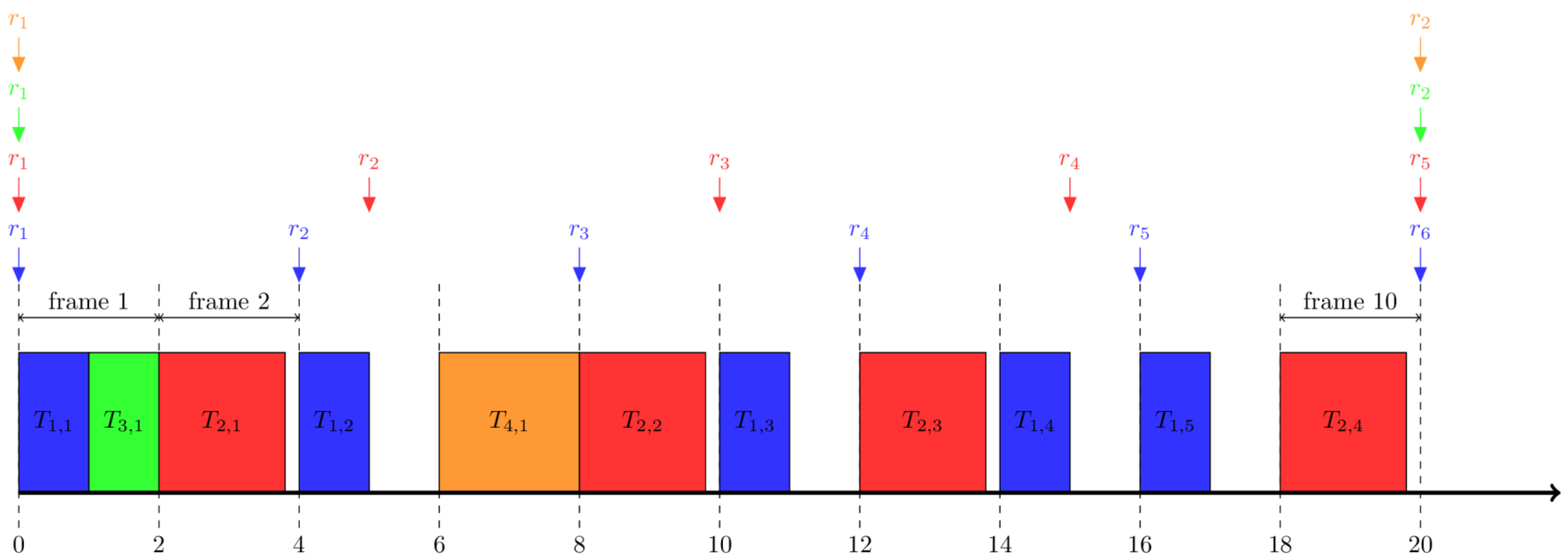
- *Example:*
  - *Tasks ($T_i = (p_i, e_i)$): T_1=(4,1), T_2=(5, 1.8), T_3=(20,1), T_4=(20,2)*
    - *Constraint 1: $f \leq 4$*
    - *Constraint 2: $f \geq 2$*
    - *Constraint 3: f = {2,4,5,10,20} → {5,10,20} can be ignored due to constraint 1*
    - *Constraint 4:*
      - *f = 2:*
        - » *T_1: 4 - GCD(4,2) = 2 $\leq$4 (ok)*
        - » *T_2: 4 - GCD(5,2) = 3 $\leq$5 (ok)*
        - » *T_3: 4 - GCD(20,2) = 2 $\leq$20 (ok)*
        - » *T_4: 4 - GCD(20,2) = 2 $\leq$20 (ok)*
      - *f = 4:*
        - » *T_1: 8 - GCD(4,4) = 4 $\leq$4 (ok)*
        - » *T_2: 8 - GCD(5,4) = 7 $\leq$5 (not ok)*

→Only feasible frame size: f = 2

Clock-Driven Scheduling

Fixed Frame Length Schedule

- *Example (continued):*
  - *Tasks (T_i=(p_i, e_i)): T_1=(4,1), T_2=(5, 1.8), T_3=(20,1), T_4=(20,2)*



Clock-Driven Scheduling

Fixed Frame Length Schedule

- Sometimes the given task set cannot meet the four frame size constraints simultaneously.
- *Example:*
  Consider the task set: $T_i=(p_i, e_i, D_i)$
  $T_1=(4,1)$, $T_2=(5,2,7)$, $T_3=(20,5)$
  - *To satisfy constraint 1:* $f \leq 4$
  - *To satisfy constraint 2:* $f \geq 5$

  → This is not possible!!!
- Solution: Partition a task into subtasks.

Clock-Driven Scheduling
Fixed Frame Length Schedule

- E.g. partitioning $T_3 = (20, 5)$ in:
  - $T_{3,1}$=(20,1),
  - $T_{3,2}$=(20,3) and
  - $T_{3,3}$=(20,1)

  yields a frame size of 4.

Clock-Driven Scheduling

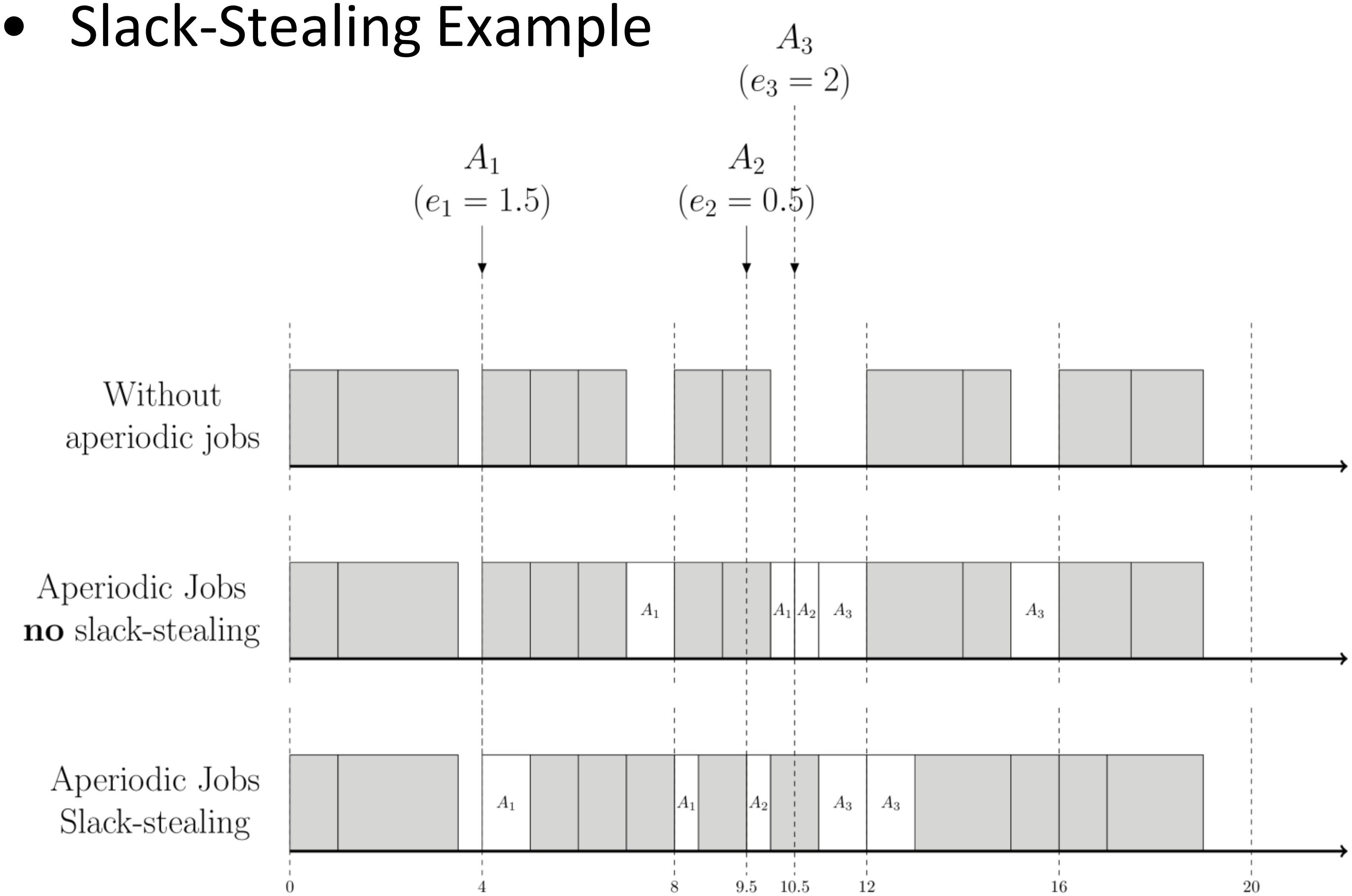
Fixed Frame Length Schedule, Aperiodic Tasks

- Aperiodic tasks are scheduled after all tasks with hard deadline requirements are scheduled.
- To improve the response time of aperiodic tasks, they should be executed ***before*** the periodic tasks.
  → ***This is called slack-stealing***

Clock-Driven Scheduling

Fixed Frame Length Schedule, Aperiodic Tasks

- Slack-Stealing Example



Average Response Time of A1, A2 and A3: 4.5

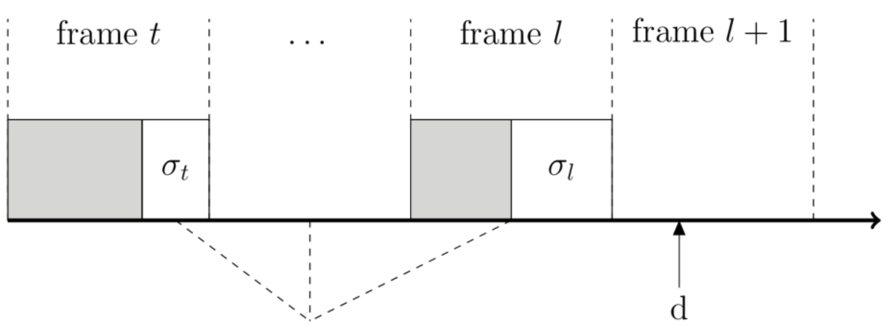
Average Response Time of A1, A2 and A3: 2.5

Clock-Driven Scheduling

Fixed Frame Length Schedule, Sporadic Tasks

- Sporadic tasks have, similar to periodic tasks, hard deadlines.
- If more than one sporadic task is waiting, they should be ordered on the Earliest-Deadline-First (EDF) basis.
- Whether a sporadic task *S (d, e)* is accepted or rejected by the scheduler is determined by an ***acceptance*** test.
  - **Acceptance Test:**
    The sporadic task *S* is accepted if the accumulated slack times from frame *t* to *l* $\sigma_c(t,l)$ is greater than or equal to the execution time of the sporadic task *S(d,e)*.

$$e \leq \sigma_c(t,l)$$



Content

Priority-Driven Scheduling

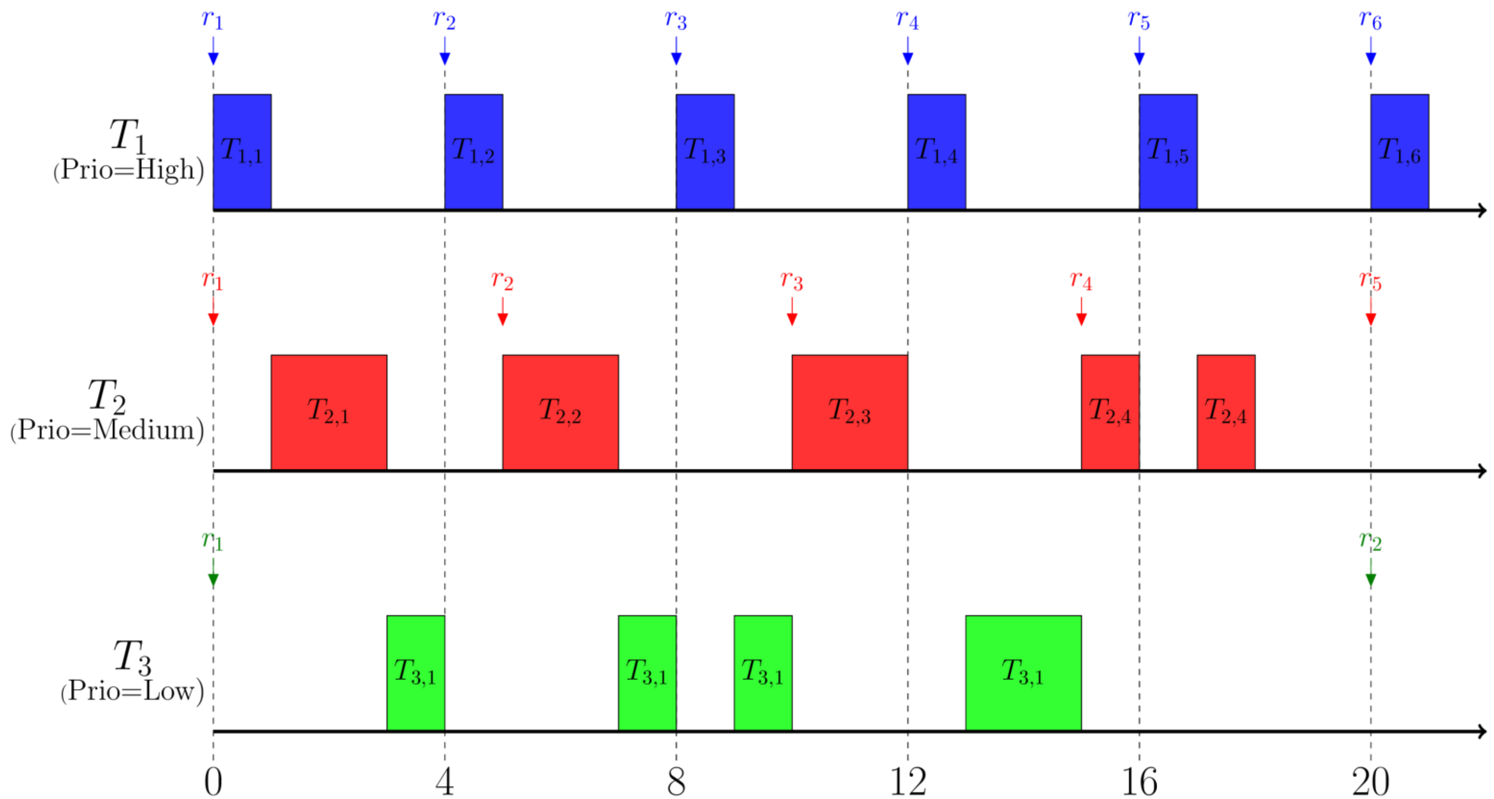
Periodic Tasks, Static Priorities, Rate Monotonic Algorithm

- In the ***rate monotonic*** (RM) algorithm, task priorities depend on the task rate ($1/p_i$)
  → the higher the rate, the higher the priority.
- *Example:*
  - *Task-Set:* $T_i = (p_i, e_i)$
    - $T_1=(4,1)$ → *Priority high*
    - $T_2=(5,2)$ → *Priority medium*
    - $T_3=(20,5)$ → *Priority low*

Priority-Driven Scheduling

Periodic Tasks, Static Priorities, Rate Monotonic Algorithm

- *Example:* $T_1=(4,1)$, $T_2=(5,2)$, $T_3=(20,5)$



Priority-Driven Scheduling

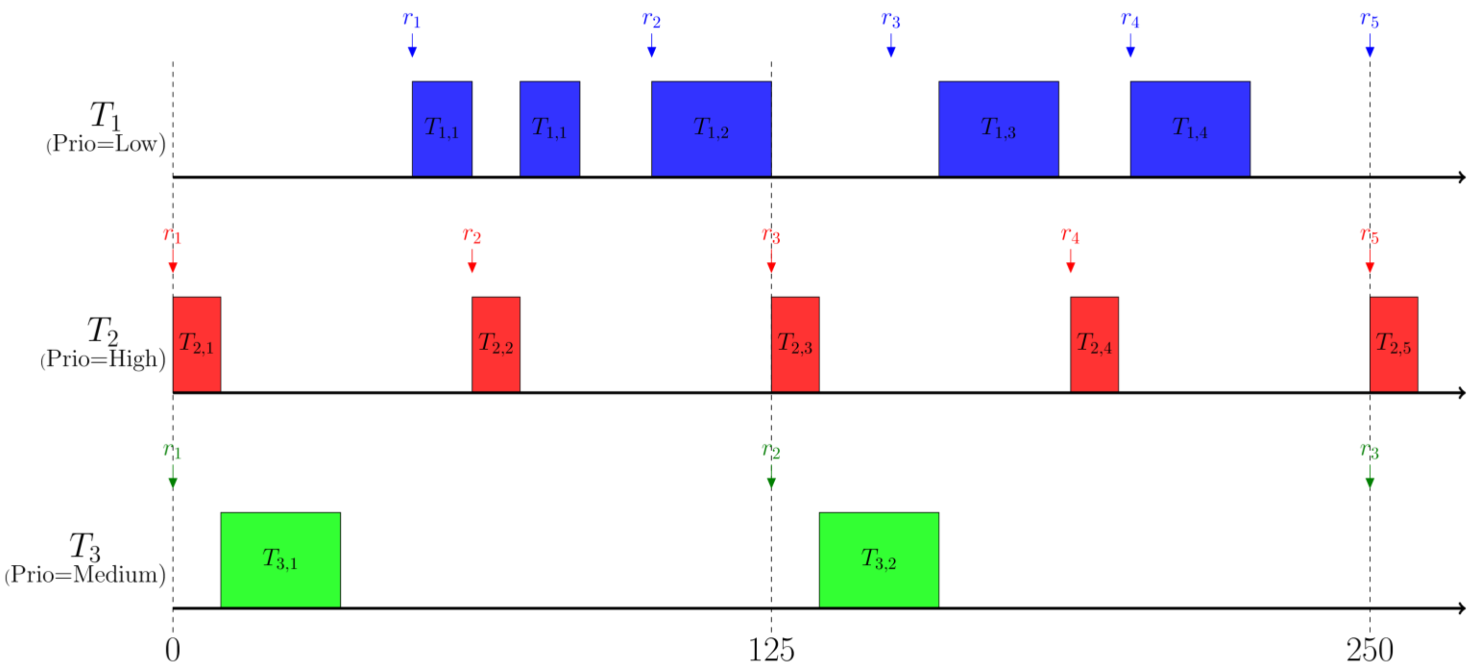
Periodic Tasks, Static Priorities, Deadline Monotonic Algorithm

- In the **deadline monotonic** (DM) algorithm, task priorities depend on the *relative* task deadline D_i
  → the shorter the relative deadline, the higher the priority.
- *Example:*
  - *Task-Set:* $T_i = (\phi_i, p_i, e_i, D_i)$
    - T_1=(50, 50, 25, 100) → *Priority low*
    - T_2=(0, 62.5, 10, 20) → *Priority high*
    - T_3=(0, 125, 25, 50) → *Priority medium*

Priority-Driven Scheduling

Periodic Tasks, Static Priorities, Deadline Monotonic Algorithm

- *Example (continued): $T_i = (\phi_i, p_i, e_i, D_i)$*
  T_1=(50, 50, 25, 100), T_2=(0, 62.5, 10, 20), T_3=(0, 125, 25, 50)



Priority-Driven Scheduling

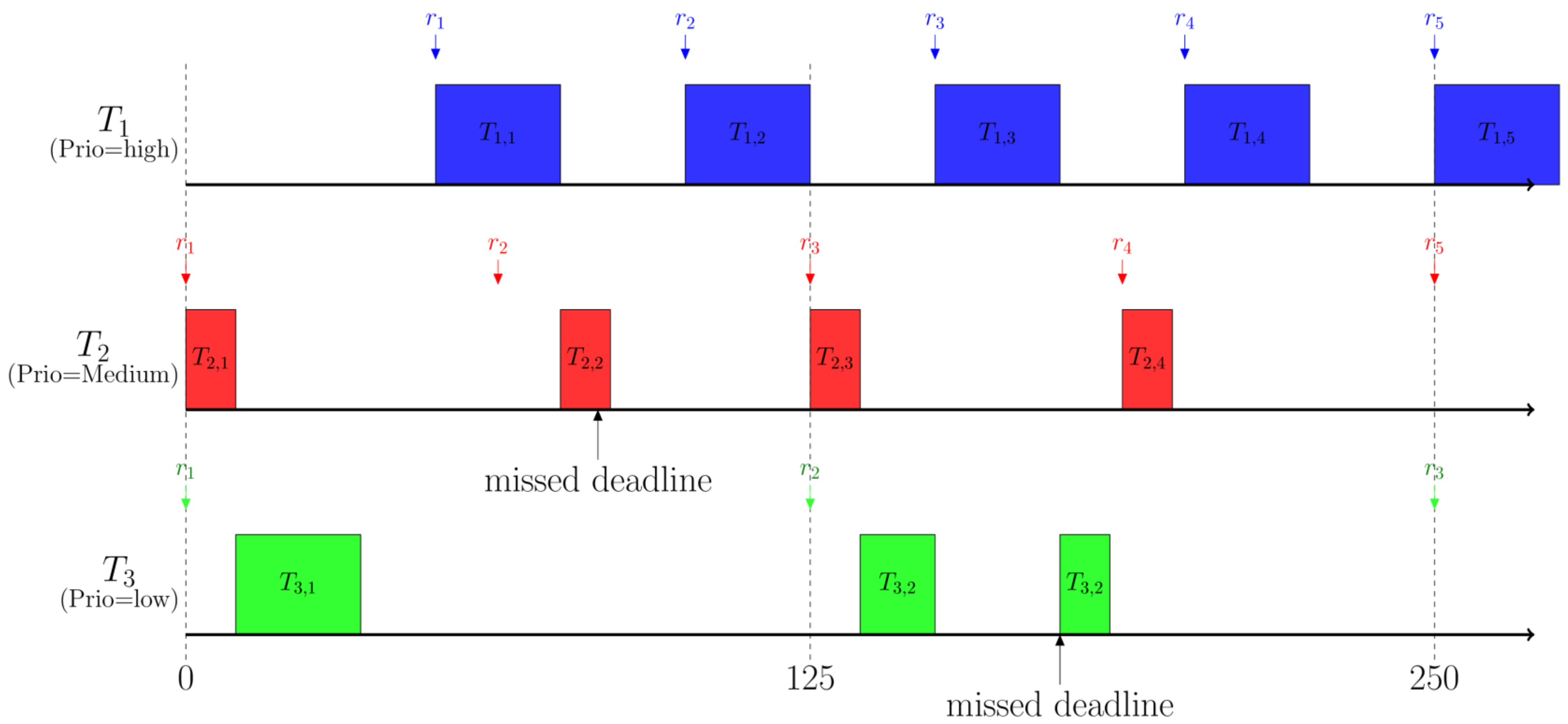
Periodic Tasks, Static Priorities, Rate vs. Deadline Monotonic

- Important notes:
  - If the relative deadlines and the periods of all tasks are proportional, the rate and deadline monotonic algorithms are identical.
  - When the relative deadlines are arbitrary, the DM algorithm can sometimes produce a feasible schedule when the RM algorithm fails.
  - The RM algorithm always fails when the DM algorithm fails.

Priority-Driven Scheduling

Periodic Tasks, Static Priorities, Rate vs. Deadline Monotonic

- Previous DM example, scheduled by a RM scheduler:
  - DM resulted in feasible schedule, RM fails.



Priority-Driven Scheduling

Periodic Tasks, Dynamic Priorities, Earliest-Deadline-First (EDF) Algorithm

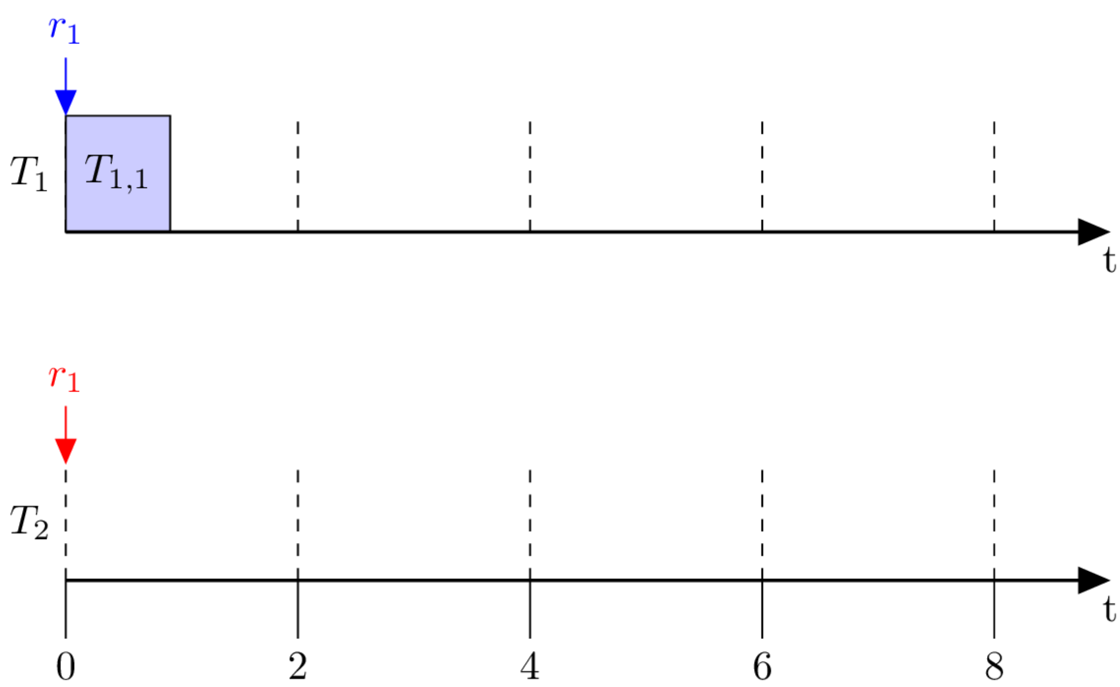
- The Earliest-Deadline-First (EDF) algorithm assigns priorities to tasks according to their **absolute** deadlines d_i.
  → The earlier the deadline, the higher the priority.
- *Example:*
  - *Given task set:* $T_i=(p_i, e_i)$
    - $T_1 = (2, 0.9)$
    - $T_2 = (5, 2.3)$

Priority-Driven Scheduling

Periodic Tasks, Dynamic Priorities, Earliest-Deadline-First (EDF) Algorithm

- *Example (continued):* $T_1 = (2, 0.9)$, $T_2 = (5, 2.3)$

t	d_i	
	T_1	T_2
0	**2**	5

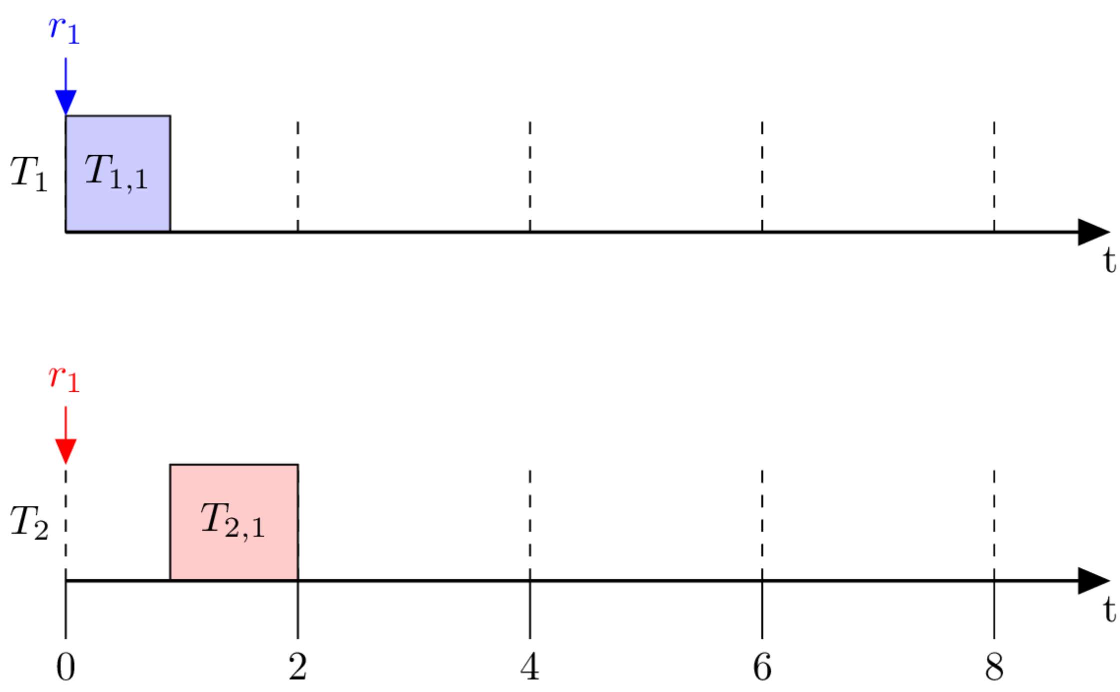


Priority-Driven Scheduling

Periodic Tasks, Dynamic Priorities, Earliest-Deadline-First (EDF) Algorithm

- *Example (continued): $T_1 = (2, 0.9)$, $T_2 = (5, 2.3)$*

t	d_i	
	T_1	T_2
0	**2**	5
0.9	-	**5**

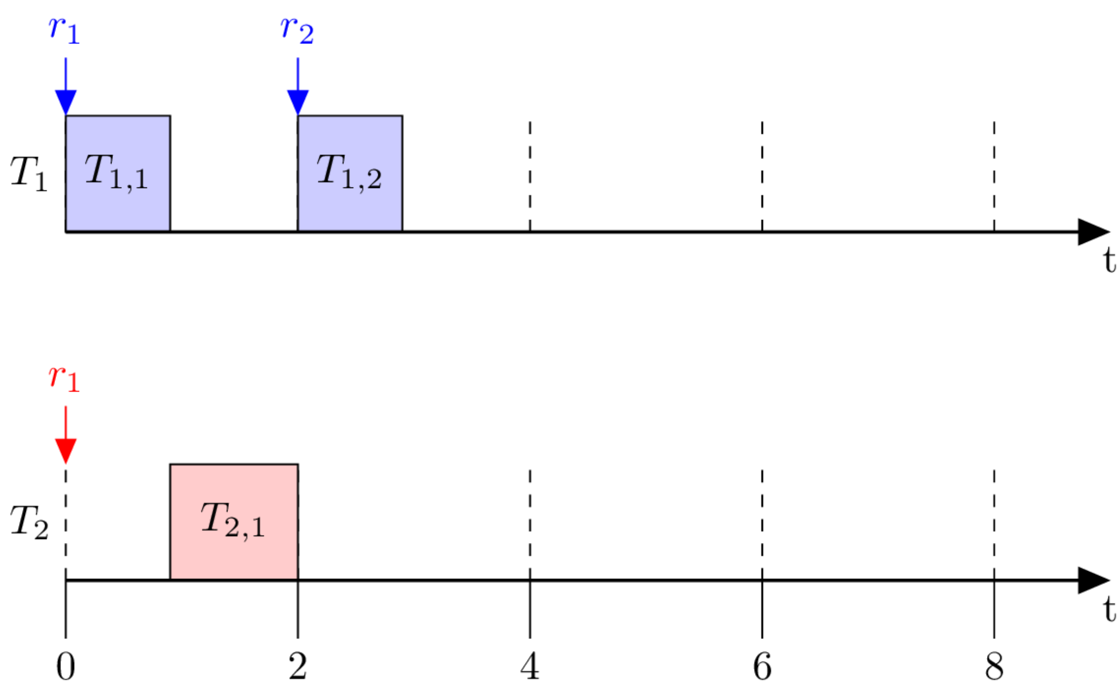


Priority-Driven Scheduling

Periodic Tasks, Dynamic Priorities, Earliest-Deadline-First (EDF) Algorithm

- *Example (continued):* $T_1 = (2, 0.9)$, $T_2 = (5, 2.3)$

t	d_i	
	T_1	T_2
0	**2**	5
0.9	-	**5**
2	**4**	5

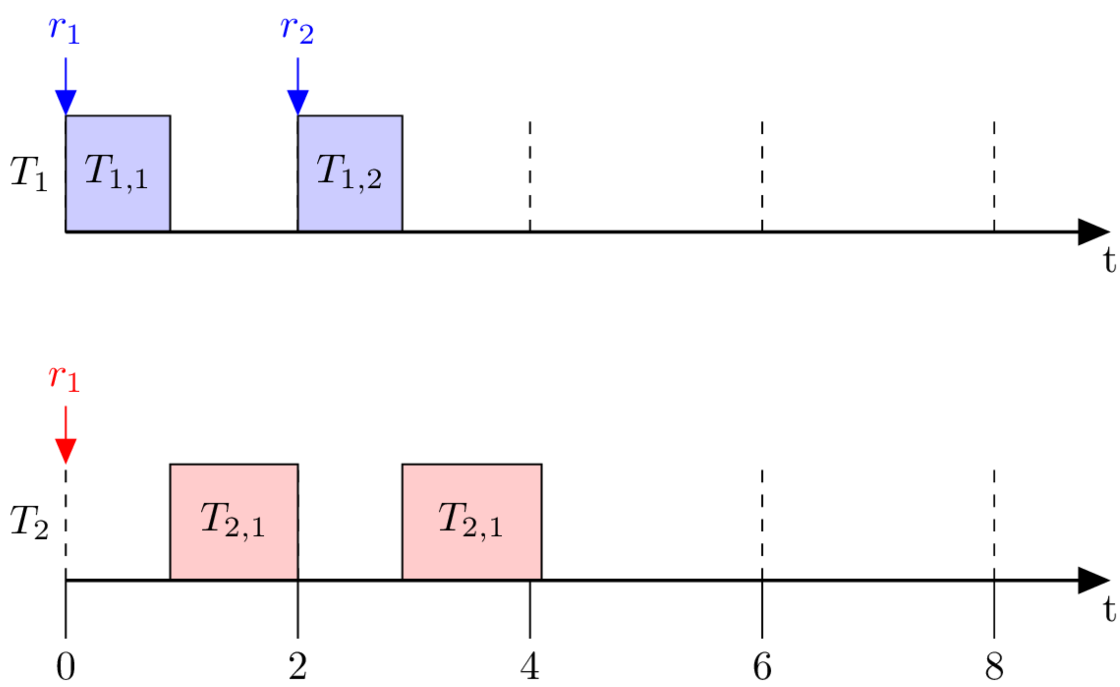


Priority-Driven Scheduling

Periodic Tasks, Dynamic Priorities, Earliest-Deadline-First (EDF) Algorithm

- *Example (continued):* $T_1 = (2, 0.9)$, $T_2 = (5, 2.3)$

t	d_i	
	T_1	T_2
0	**2**	5
0.9	-	**5**
2	**4**	5
2.9	-	**5**

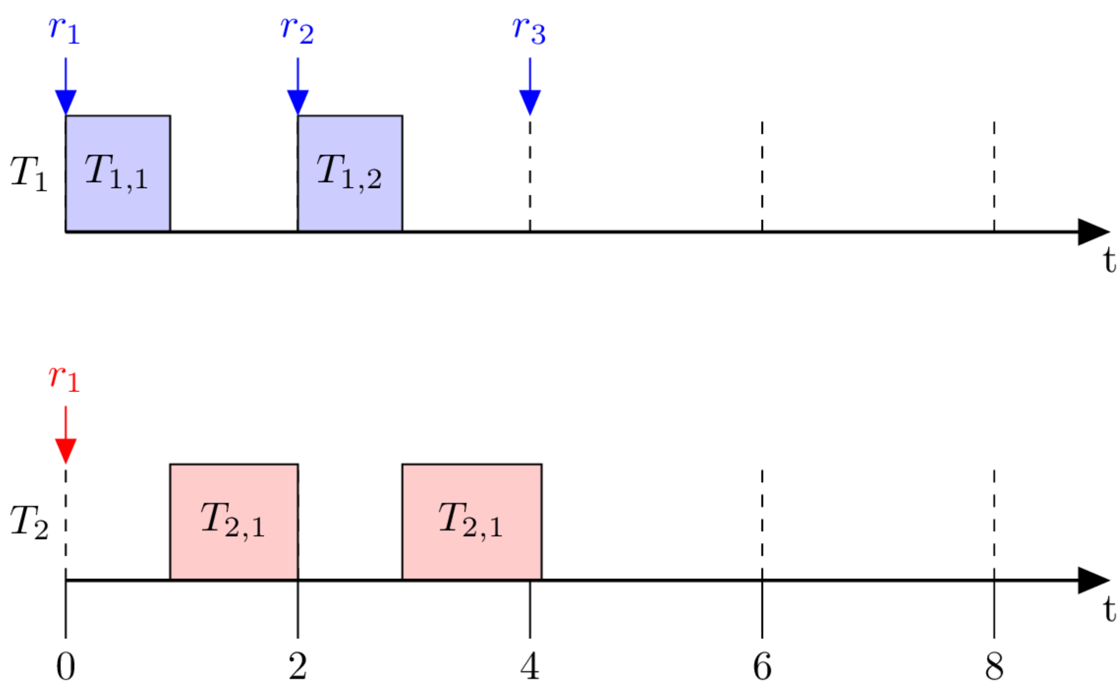


Priority-Driven Scheduling

Periodic Tasks, Dynamic Priorities, Earliest-Deadline-First (EDF) Algorithm

- *Example (continued):* $T_1 = (2, 0.9)$, $T_2 = (5, 2.3)$

t	d_i	
	T_1	T_2
0	**2**	5
0.9	-	**5**
2	**4**	5
2.9	-	**5**
4	6	**5**

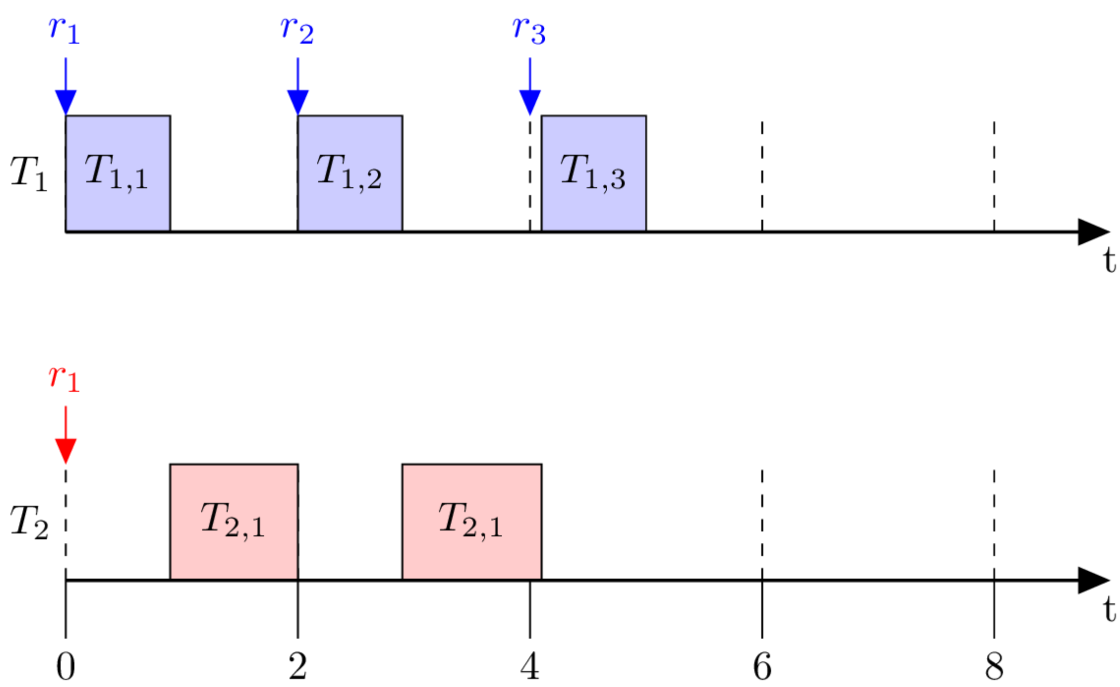


Priority-Driven Scheduling

Periodic Tasks, Dynamic Priorities, Earliest-Deadline-First (EDF) Algorithm

- *Example (continued):* $T_1 = (2, 0.9)$, $T_2 = (5, 2.3)$

t	d_i	
	T_1	T_2
0	**2**	5
0.9	-	**5**
2	**4**	5
2.9	-	**5**
4	6	**5**
4.1	**6**	-

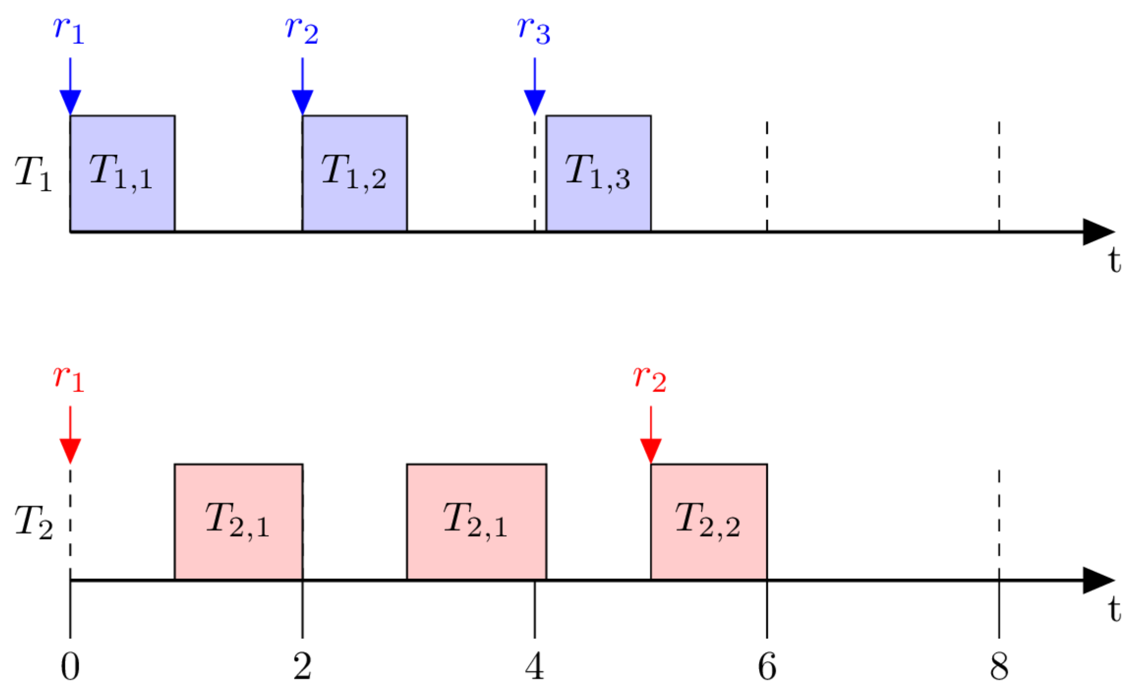


Priority-Driven Scheduling

Periodic Tasks, Dynamic Priorities, Earliest-Deadline-First (EDF) Algorithm

- *Example (continued):* $T_1 = (2, 0.9)$, $T_2 = (5, 2.3)$

t	d_i	
	T_1	T_2
0	**2**	5
0.9	-	**5**
2	**4**	5
2.9	-	**5**
4	6	**5**
4.1	**6**	-
5	-	**10**

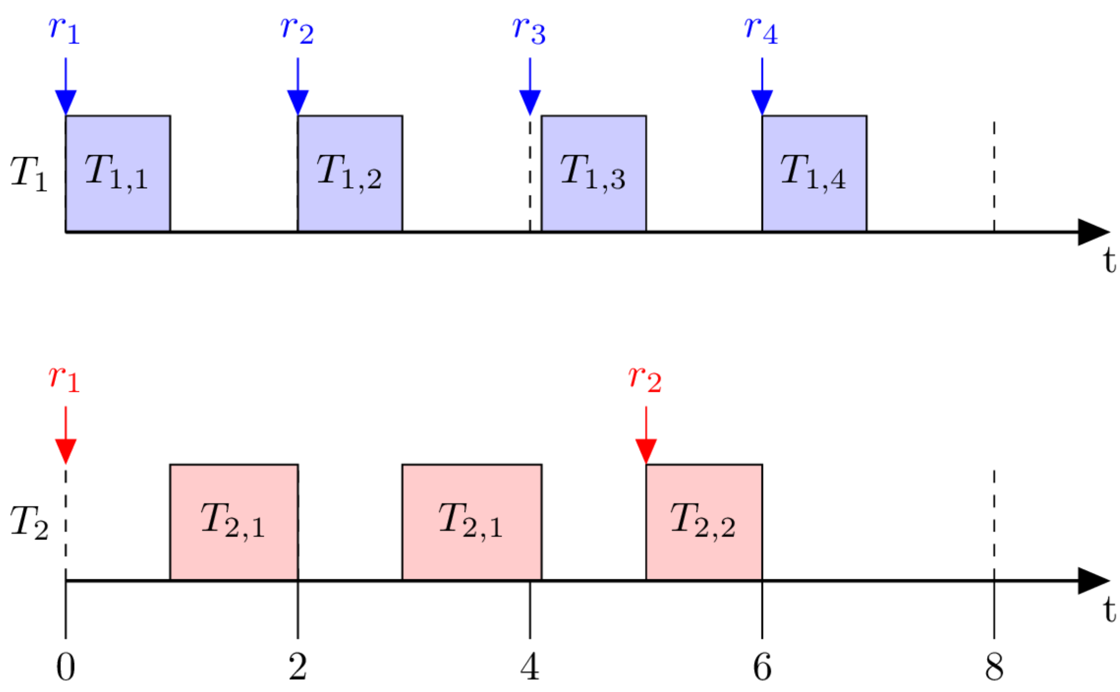


Priority-Driven Scheduling

Periodic Tasks, Dynamic Priorities, Earliest-Deadline-First (EDF) Algorithm

- *Example (continued):* $T_1 = (2, 0.9)$, $T_2 = (5, 2.3)$

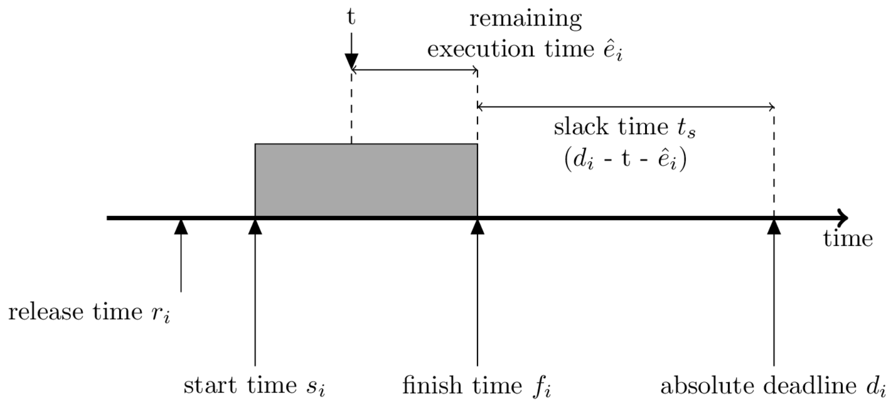
t	d_i	
	T_1	T_2
0	**2**	5
0.9	-	**5**
2	**4**	5
2.9	-	**5**
4	6	**5**
4.1	**6**	-
5	-	**10**
6	**8**	10



Priority-Driven Scheduling

Periodic Tasks, Dynamic Priorities, Least-Slack-Time-First (LST) Algorithm

- The Least-Slack-Time-First algorithm assigns priorities to tasks according to their **slack time**.
  → the smaller the slack time, the higher the priority
- Definition of slack time (recapitulation):



Note:

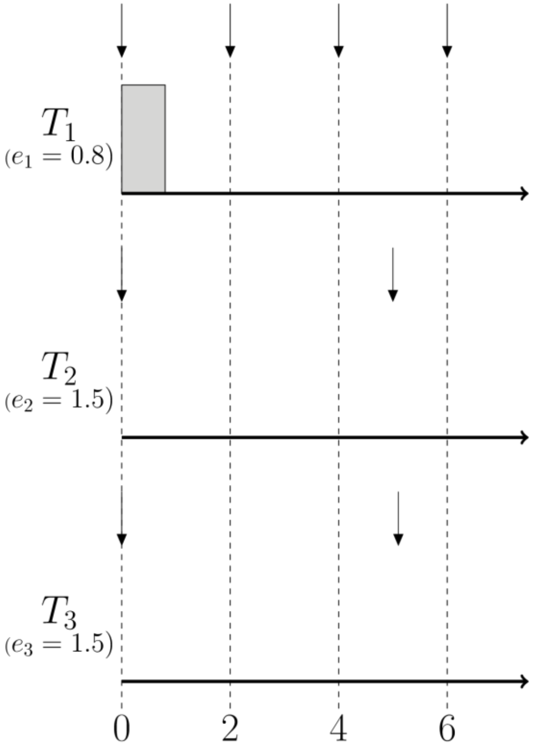
- Slack time of currently running processes is constant.
- Slack time of waiting processes shortens.

Priority-Driven Scheduling

Periodic Tasks, Dynamic Priorities, Least-Slack-Time-First (LST) Algorithm

- *Example (T_i=(p_i, e_i)): T_1 = (2, 0.8), T_2 = (5, 1.5), T_3 = (5.1, 1.5)*
- *Slack-Time:* $t_s = d - t - \hat{e}$

t	d / ê / t_s		
	T_1	T_2	T_3
0	**2 / 0.8 / 1.2**	5 / 1.5 / 3.5	5.1 / 1.5 / 3.6

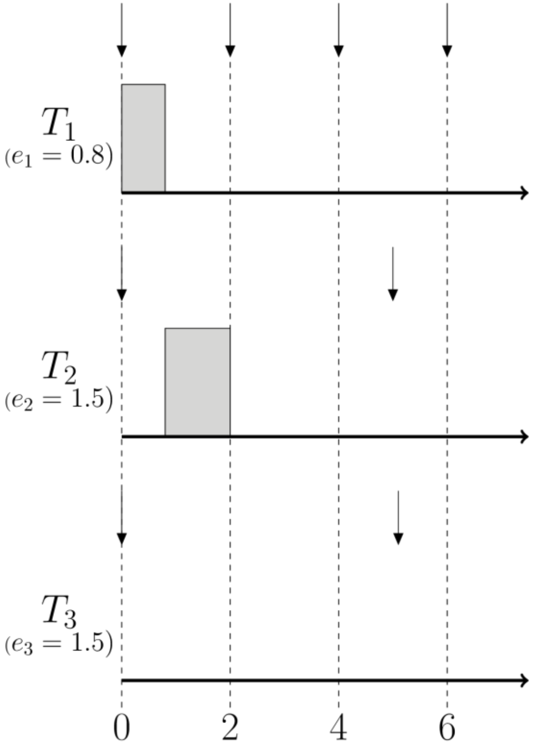


Priority-Driven Scheduling

Periodic Tasks, Dynamic Priorities, Least-Slack-Time-First (LST) Algorithm

- *Example ($T_i=(p_i, e_i)$): T_1 = (2, 0.8), T_2 = (5, 1.5), T_3 = (5.1, 1.5)*
- *Slack-Time:* $t_s = d - t - \hat{e}$

t	d / ê / t_s		
	T_1	T_2	T_3
0	**2 / 0.8 / 1.2**	5 / 1.5 / 3.5	5.1 / 1.5 / 3.6
0.8	-	**5 / 1.5 / 2.7**	5.1 / 1.5 / 2.8

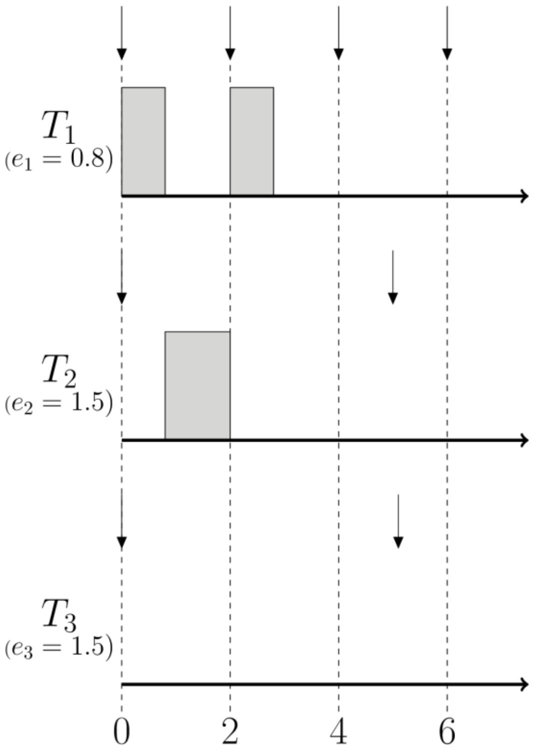


Priority-Driven Scheduling

Periodic Tasks, Dynamic Priorities, Least-Slack-Time-First (LST) Algorithm

- *Example ($T_i=(p_i, e_i)$): T_1 = (2, 0.8), T_2 = (5, 1.5), T_3 = (5.1, 1.5)*
- *Slack-Time:* $t_s = d - t - \hat{e}$

t	d / ê / t_s		
	T_1	T_2	T_3
0	**2 / 0.8 / 1.2**	5 / 1.5 / 3.5	5.1 / 1.5 / 3.6
0.8	-	**5 / 1.5 / 2.7**	5.1 / 1.5 / 2.8
2	**4 / 0.8 / 1.2**	5 / 0.3 / 2.7	5.1 / 1.5 / 1.6

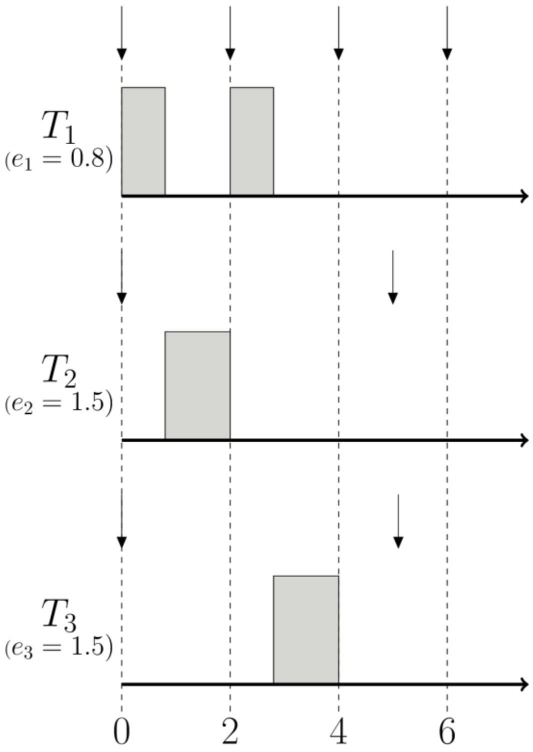


Priority-Driven Scheduling

Periodic Tasks, Dynamic Priorities, Least-Slack-Time-First (LST) Algorithm

- *Example ($T_i=(p_i, e_i)$): T_1 = (2, 0.8), T_2 = (5, 1.5), T_3 = (5.1, 1.5)*
- *Slack-Time:* $t_s = d - t - \hat{e}$

t	d / ê / t_s		
	T_1	T_2	T_3
0	**2 / 0.8 / 1.2**	5 / 1.5 / 3.5	5.1 / 1.5 / 3.6
0.8	-	**5 / 1.5 / 2.7**	5.1 / 1.5 / 2.8
2	**4 / 0.8 / 1.2**	5 / 0.3 / 2.7	5.1 / 1.5 / 1.6
2.8	-	5 / 0.3 / 1.9	**5.1 / 1.5 / 0.8**

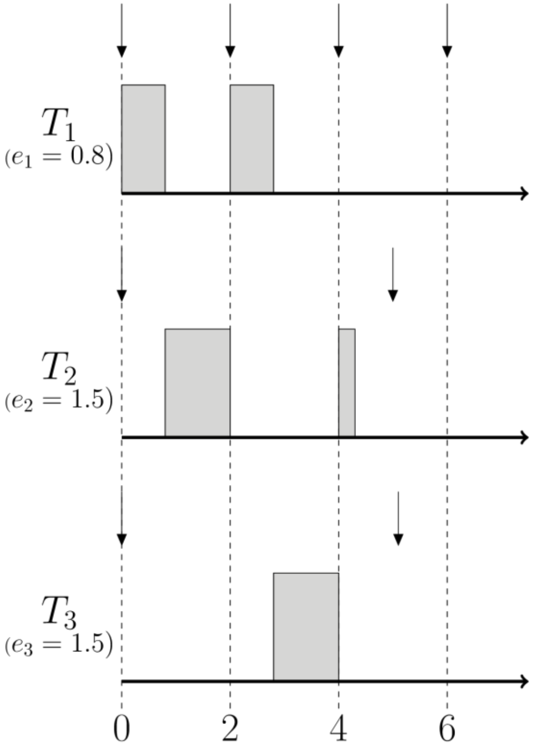


Priority-Driven Scheduling

Periodic Tasks, Dynamic Priorities, Least-Slack-Time-First (LST) Algorithm

- *Example ($T_i=(p_i, e_i)$): T_1 = (2, 0.8), T_2 = (5, 1.5), T_3 = (5.1, 1.5)*
- *Slack-Time:* $t_s = d - t - \hat{e}$

t	d / ê / t_s		
	T_1	T_2	T_3
0	**2 / 0.8 / 1.2**	5 / 1.5 / 3.5	5.1 / 1.5 / 3.6
0.8	-	**5 / 1.5 / 2.7**	5.1 / 1.5 / 2.8
2	**4 / 0.8 / 1.2**	5 / 0.3 / 2.7	5.1 / 1.5 / 1.6
2.8	-	5 / 0.3 / 1.9	**5.1 / 1.5 / 0.8**
4	6 / 0.8 / 1.2	**5 / 0.3 / 0.7**	5.1 / 0.3 / 0.8



Priority-Driven Scheduling

Periodic Tasks, Dynamic Priorities, Summary EDF and LST

- Both, EDF and LST are optimal if:
  - Preemption of tasks is allowed
  - Tasks do not contend for resources
  - A single processor system is used
- EDF does not require knowledge of execution times, LST does
  → huge drawback

Content

Schedulability Testing

Introduction

- A test to validate that a given set of tasks can meet its hard deadlines when scheduled according to a specific scheduling algorithm is called ***schedulability*** test.

Schedulability Testing

DM and RM Algorithms

- A task set of *n* tasks can be *feasibly* scheduled on *one processor* by the RM algorithm if the following utilization condition holds (Liu und Layland 1973):

$$U = \sum_{i=1}^{n} \frac{e_i}{p_i} \leq n(2^{1/n} - 1)$$

- Note: The tasks have to be:
  - independent,
  - preemptable, and
  - periodic.

Recapitulation: If the relative deadlines of all task in a given task set are proportional to the periods, the DM algorithm is identical to the RM algorithm and the above condition can also be used to perform a schedulability test for the DM algorithm.

Schedulability Testing

DM and RM Algorithms

- *Example:*

Task	p_i	e_i	u_i
1	1.0	0.25	0.25
2	1.25	0.1	0.08
3	1.5	0.3	0.2
4	1.75	0.07	0.04
5	2.0	0.1	0.05
			Sum: 0.62

Total utilization $U=0.62 \leq 0.743$ → task set can be feasibly scheduled by the RM algorithm.

Schedulability Testing

DM and RM Algorithms

- *Important:*
  The presented condition is not a necessary condition !!!
  → Even if the utilization of a task set exceeds the condition, a feasible RM schedule might exist.
- A schedulability test of such a task set, scheduled by a fixed-priority algorithm, can be performed by the **time-demand analysis**.

Schedulability Testing

Time-Demand Analysis for Fixed-Priority Algorithms

- For a sorted task set T_i (i.e. T_0 = task with highest priority, T_i = task with lowest priority), we can perform a time-demand analysis, by (Lehoczky et al., 1989)
  1. computing the time-demand of all tasks T_i, according to:

     $$w_i(t) = e_i + \sum_{k=1}^{i-1} \left\lceil \frac{t}{p_k} \right\rceil e_k \quad \text{for } 0 < t \le p_i$$

  2. checking whether the inequality

     $$w_i(t) \le t$$

     is satisfied for values of *t* that are equal to

     $$t = jp_k; k = 1, 2, \ldots, i; j = 1, 2, \ldots, \lfloor \min(p_i, D_i) / p_k \rfloor$$

     If this inequality is satisfied at one of these instants, T_i is schedulable.

Schedulability Testing

Time-Demand Analysis for Fixed-Priority Algorithms

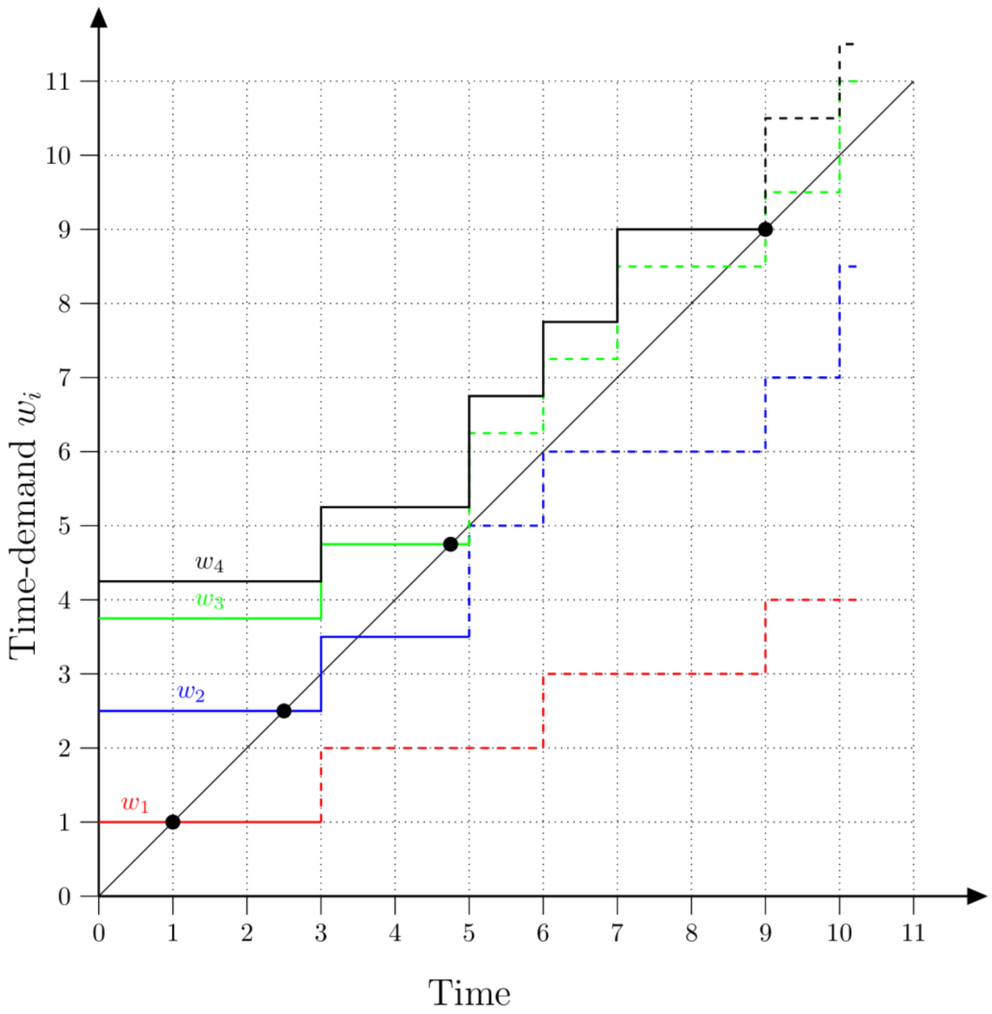
- *Example:*
  $T_1=(\phi_1, 3, 1); T_2=(\phi_2, 5, 1.5), T_3=(\phi_3, 7, 1.25), T_4=(\phi_4, 9, 0.5)$
  - w_1:
    - $w_1(3) = 1 \leq 3 \rightarrow$ *OK*
  - w_2:
    - $w_2(3) = 1.5 + 1 = 2.5 \leq 3 \rightarrow$ *OK*
  - w_3:
    - $w_3(3) = 1.25 + 1 + 1.5 = 3.75 > 3 \rightarrow$ *Not OK*
    - $w_3(5) = 1.25 + 2 + 1.5 = 4.75 \leq 5 \rightarrow$ *OK*
  - w_4:
    - $w_4(3) = 0.5 + 1 + 1.5 + 1.25 = 4.25 > 3 \rightarrow$ *Not OK*
    - $w_4(5) = 0.5 + 2 + 1.5 + 1.25 = 5.25 > 5 \rightarrow$ *Not OK*
    - $w_4(6) = 0.5 + 2 + 3 + 1.25 = 6.75 > 6 \rightarrow$ *Not OK*
    - $w_4(7) = 0.5 + 3 + 3 + 1.25 = 7.75 > 7 \rightarrow$ *Not OK*
    - $w_5(9) = 0.5 + 3 + 3 + 2.5 = 9 \leq 9 \rightarrow$ *OK*

Schedulability Testing

Time-Demand Analysis for Fixed-Priority Algorithms

- *Example (continued):*

  Graphical demonstration of time-demand analysis



Schedulability Testing

EDF Algorithm

- Task density:

$$\text{density}_k = \frac{e_k}{\min(D_k, p_k)}$$

- A set of
  - independent,
  - periodic, and
  - preemptable

  tasks can be *feasibly* scheduled by the EDF algorithm on one processor if the task set density is less or equal to 1:

$$\sum_{k=1}^{n} \frac{e_k}{\min(D_k, p_k)} \leq 1$$

Note: This is only a sufficient condition. Even if inequality is not satisfied, a feasible schedule might exist.

Content

Resources and Resource Access Control

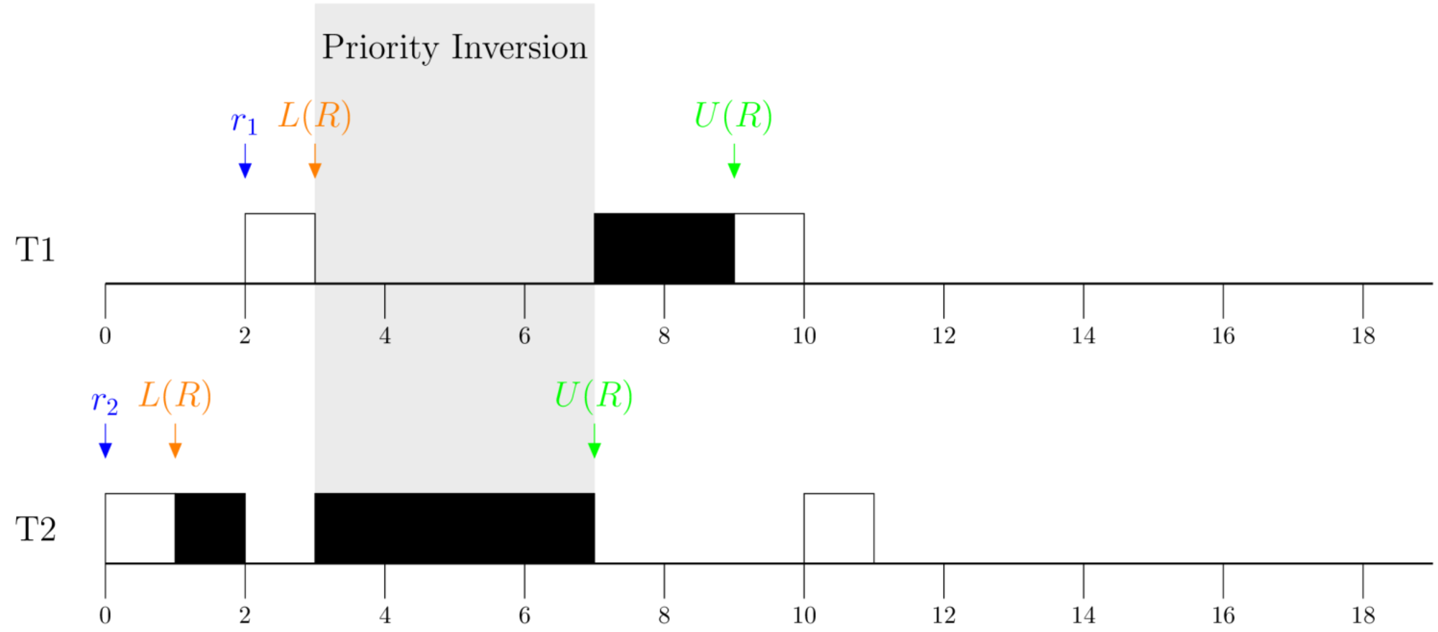
Introduction

- If resources can only be used in a mutual exclusive manner, resource contentions occur that can lead to system failures.
- Effects of resource contentions:
  - Priority Inversions
  - Deadlocks

Resources and Resource Access Control

Effects of Resource Contention: Priority Inversion

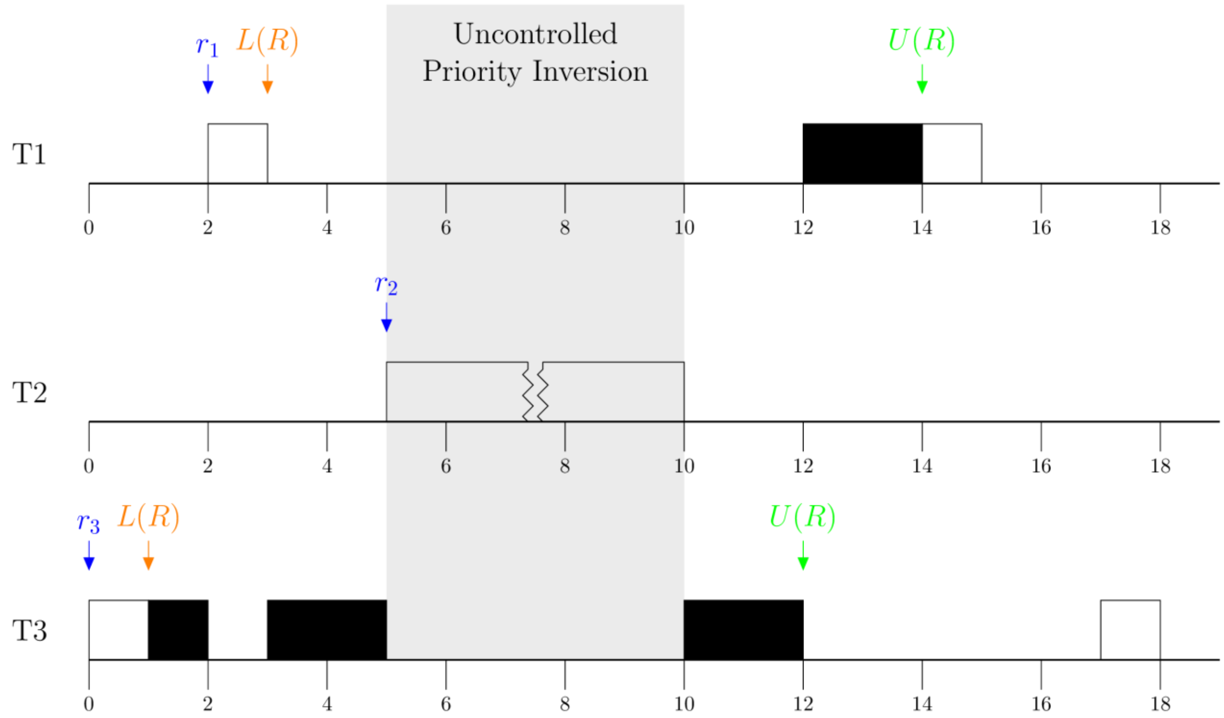
- The phenomenon that a lower-priority task blocks a higher-priority task is called ***priority inversion***.



Resources and Resource Access Control

Effects of Resource Contention: Uncontrolled Priority Inversion

- **Uncontrolled (or Unbounded) Priority Inversion**
  A medium priority task can block a high priority task forever.



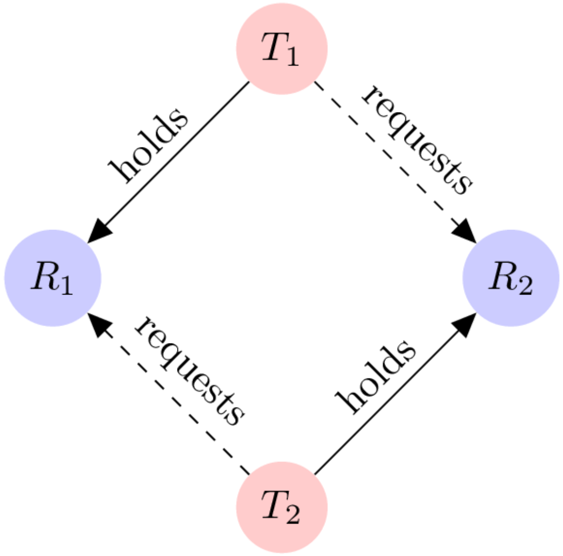
Uncontrolled priority inversion can only occur if the task set contains more than 2 tasks.

Resources and Resource Access Control

Effects of Resource Contention: Deadlock

- Consider two tasks T_1 and T_2 and two resources R_1 and R_2.
  - T1 holds R1, requests R2
  - T2 holds R2, requests R1

→ Deadlock



Resources and Resource Access Control

Nonpreemptive Critical Section (NPCS) Protocol

- Simple way to control access to a resource is to schedule all critical sections nonpreemptively:

  If a task request a resource, it is always allocated the resource and executes with the highest priority.

  → This protocol is called the Nonpreemptive Critical Section (NPCS) protocol

- As no preemption takes place, no deadlock or priority inversion can occur!!!
- Shortcoming: Every task can be blocked by every lower-priority task, even if there is no resource conflict.

Resources and Resource Access Control

Basic Priority Inheritance Protocol (BPIP)

- The basic priority inheritance protocol (BPIP) prevents uncontrolled priority inversions but not deadlocks.

  → This is achieved by raising the ***current*** *priority* $\pi_l(t)$ of a lower-priority task to a higher (*inherited*) priority $\pi_h(t)$ of another task.

- BPIP rules:
  - *Scheduling Rule*: Ready tasks are scheduled preemptively in a priority-driven manner according to their ***current*** priorities. At the release time, the current priority $\pi(t)$ is equal to the assigned priority (the priority determined by the scheduling algorithm).

Resources and Resource Access Control

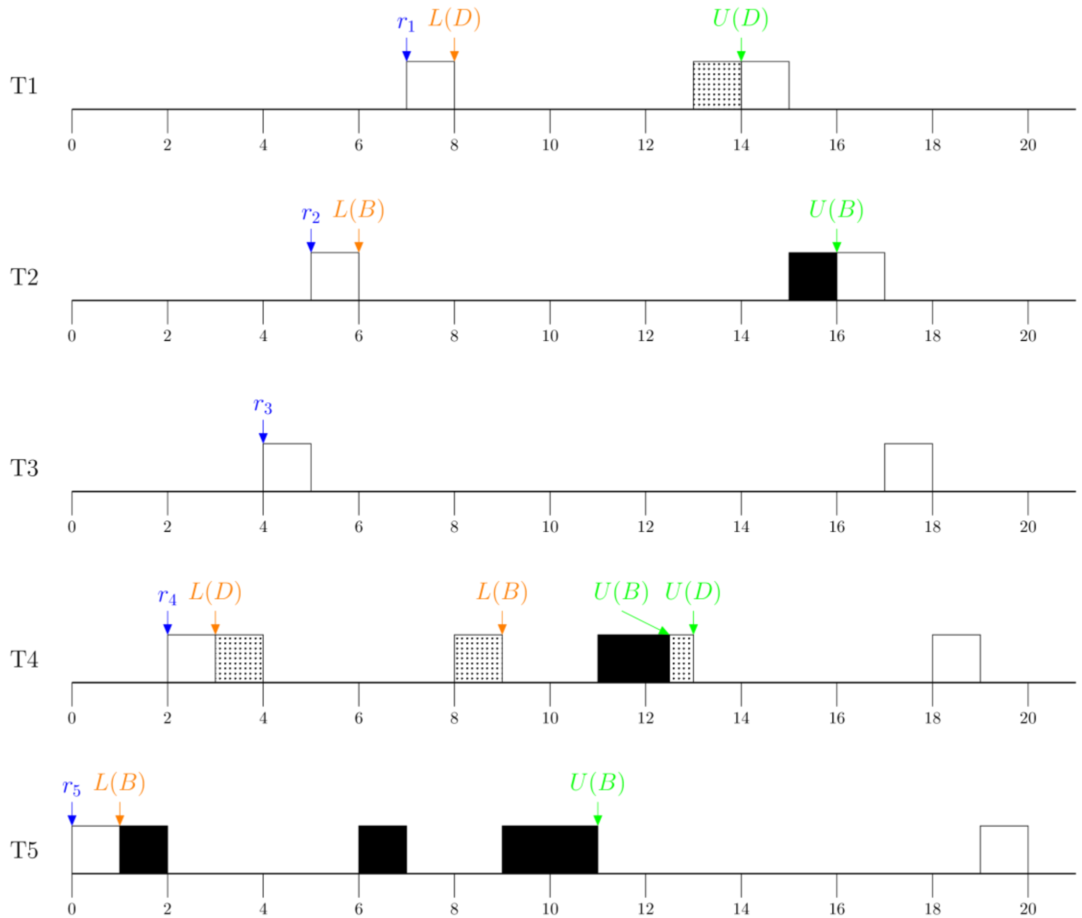
Basic Priority Inheritance Protocol (BPIP)

- BPIP rules (continued):
  - *Allocation Rule*: When a task *T* requests a resource *R* at time *t*,
    a) if *R* is free, *R* is allocated to *T* until *T* releases the resource, and
    b) if *R* is not free, the request is denied and *T* is blocked.
  - *Priority-Inheritance Rule*: When the requesting task *T* becomes blocked, the task T_l which blocks *T* inherits the current priority of *T* until it releases the resource. At that time, the priority of T_l returns to the value it had at the time when it acquired *R*.

Resources and Resource Access Control

Basic Priority Inheritance Protocol (BPIP), Example

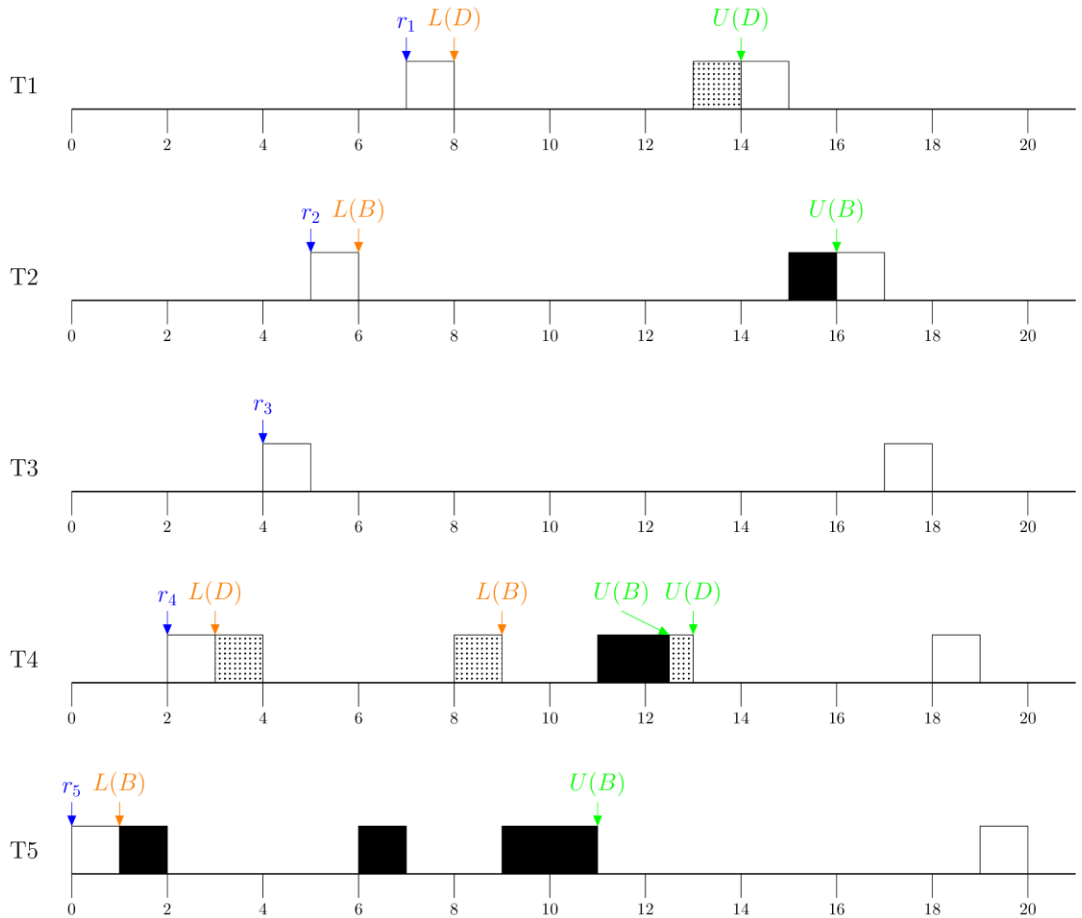
Time	Event
0	T5 executes with priority 5
1	T5 is granted resource “black”
2	T4 released, preempts T5
3	T4 is granted resource “dotted”
4	T3 released, preempts T4
5	T2 released, preempts T3
6	T2 requests resource “black”, T5 inherits priority of T2 and executes
7	T1 released, preempts T5



Resources and Resource Access Control

Basic Priority Inheritance Protocol (BPIP), Example

Time	Event
8	T1 requests resource “dotted”, T4 inherits priority of T1
9	T4 requests resource “black”, T5 inherits priority and executes
11	T5 releases resource “black”, T4 continues
13	T4 releases resource “dotted”, T1 acquires resource “dotted” and continues
15	T1 completes, T2 is granted resource “black” and executes
17	T2 completes, afterwards T3, T4 and T5 execute and complete



Resources and Resource Access Control

Basic Priority Ceiling Protocol (BPCP)

- The basic priority ceiling protocol (BPCP) extends the BPIP to prevent deadlocks and to further reduce the blocking time.
- **Priority Ceiling**: The priority ceiling $\Pi(R_i)$ of a resource R_i is the highest priority of all the tasks that require R_i.
  - *Example (based on previous slide): $\Pi(B) = 2$, $\Pi(D) = 1$*
- **Current Priority Ceiling (or simply *ceiling*):** The ceiling $\hat{\Pi}(t)$ is equal to the highest priority ceiling of the resources currently in use. If all resources are free, the ceiling is equal to Ω, a non-existing priority lower than any other priority.
  - Example (based on previous slide):
    - In (1,3], resource „black“ is used; hence the ceiling is 2
    - In (3,13], resource „dotted“ is used; hence the ceiling is 1

Resources and Resource Access Control

Basic Priority Ceiling Protocol (BPCP)

- BPCP rules:
  - *Scheduling Rule:*
    - a) At its release time, the current task priority $\pi(t)$ is equal to its assigned priority.
    - b) Every ready task is scheduled preemptively and in a priority-driven manner, depending on its current priority $\pi(t)$.
  - *Allocation rule:*
    Whenever a task *T* requests a resource *R* at time *t*, one of the following conditions occurs:
    - *a)* *R* is held by another task → *T* blocks
    - *b)* *R* is free
      - a) If the priority $\pi(t)$ of *T* is higher than the current priority ceiling, *R* is allocated to *T*.
      - b) If the priority of *T* is **not** higher than the ceiling, *R* is allocated to *T* only if *T* is holding the resource whose priority ceiling is equal to the ceiling; otherwise *T* blocks.

Resources and Resource Access Control

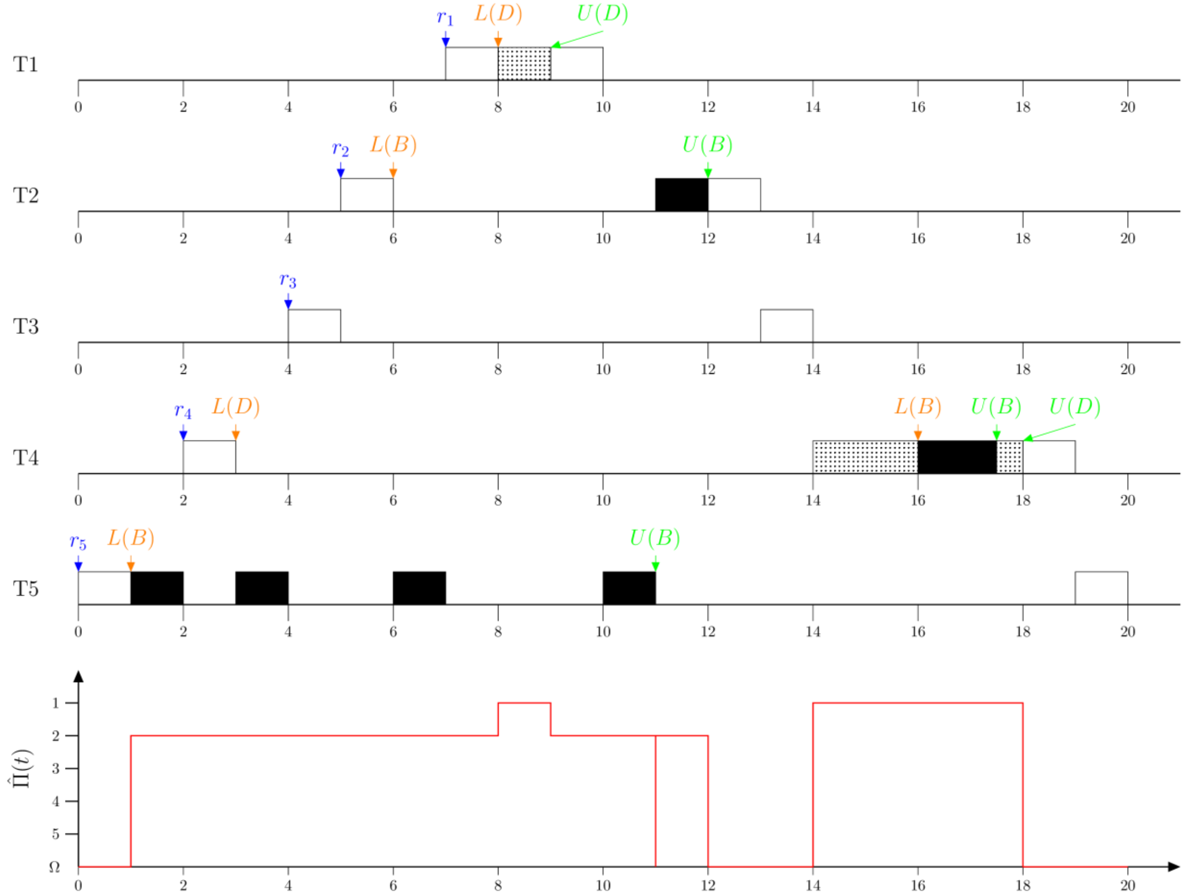
Basic Priority Ceiling Protocol (BPCP)

- BPCP rules:
  - *Priority Inheritance Rule*: When *T* becomes blocked, the task T_l that blocks *T* inherits the current priority of T. T_l executes at its inherited priority until the time when it releases every resource whose priority ceiling is equal to or higher than the priority of T; at that time, the priority of T_l returns to the value it had when it was granted the resource.

Resources and Resource Access Control

Basic Priority Ceiling Protocol (BPCP), Example

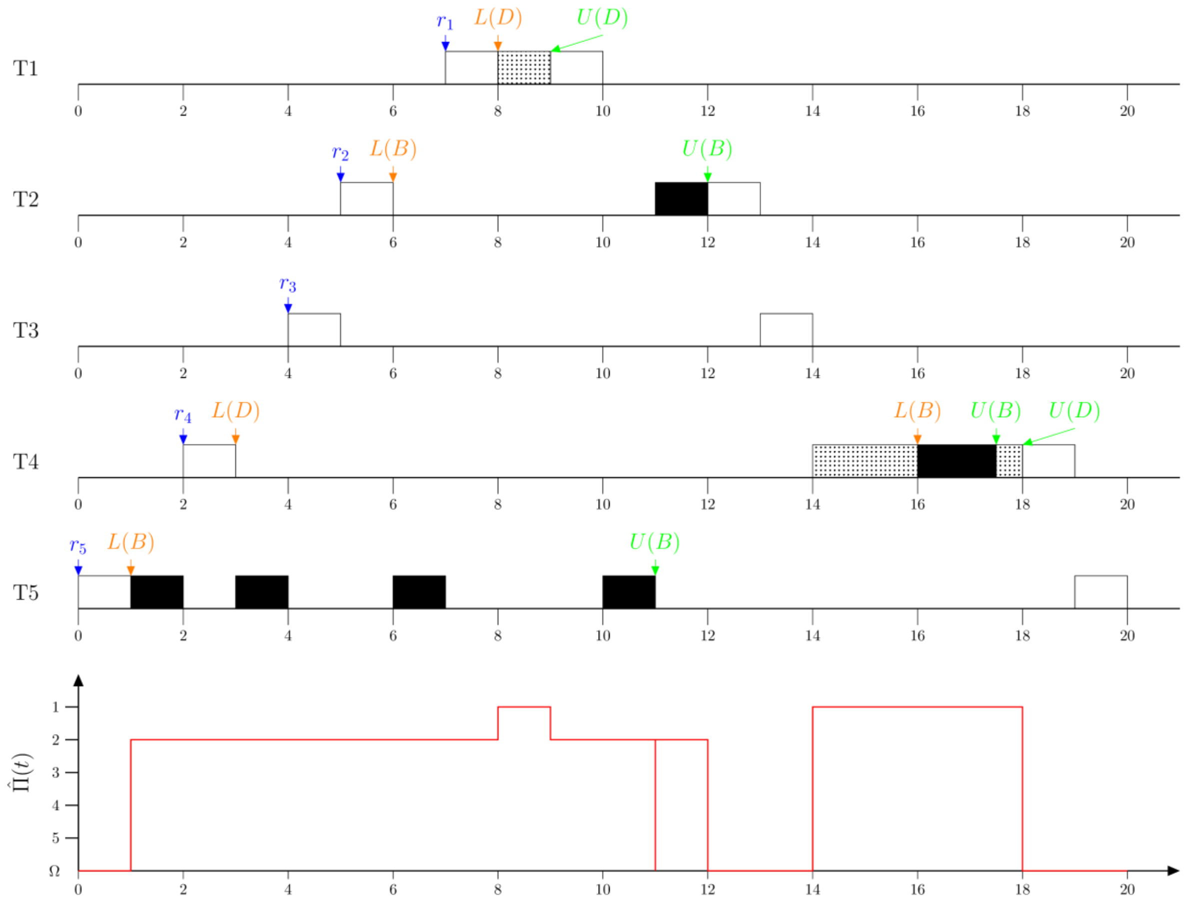
Time	Event
0	T5 executes with priority 5
1	T5 is granted resource “black”
2	T4 released, preempts T5
3	T4 requests resource “dotted”, but the request is denied (priority of T4 lower than current ceiling). T5 inherits priority of T4 and executes at priority 4.
4	T3 released, preempts T5
5	T2 released, preempts T3
6	T2 requests resource “black” and becomes blocked by T5; T5 inherits priority 2



Resources and Resource Access Control

Basic Priority Ceiling Protocol (BPCP), Example

Time	Event
7	T1 becomes ready and preempts T5
8	T1 requests resource “dotted”; Priority of T1 higher than ceiling, resource request is granted
10	T3 and T5 are ready, T5 has higher priority (2) and executes
11	T5 releases “black” and its priority returns to 5; the ceiling drops to Ω; T2 unblocks, allocates „black“ and executes
14	J4 is granted “dotted” as its priority is higher than the ceiling



Resources and Resource Access Control

Basic Priority Ceiling Protocol (BPCP), Example

Time	Event
16	T4 requests “black”, which is free. The priority of T4 is lower than the ceiling, but T4 is holding the resource whose priority ceiling is equal to the current ceiling (“dotted”).