Embedded Systems Programming – Real-Time Scheduling 1

Raimund Kirner
University of Hertfordshire

acknowledgement: includes slides by Peter Puschner

Overview

- Real-time Computing
- Static Priority Scheduling
- Cyclic Executive
- Rate-monotonic Scheduling (RMS)
- Response Time Analysis for RMS

The Challenge

- System consists of multiple tasks, to be executed in a periodic manner
- Each task T_X consists of
  - period (p_X)
  - execution time (c_X)
  - implicit relative deadline (= period)

task T_X
(p_X,c_X)

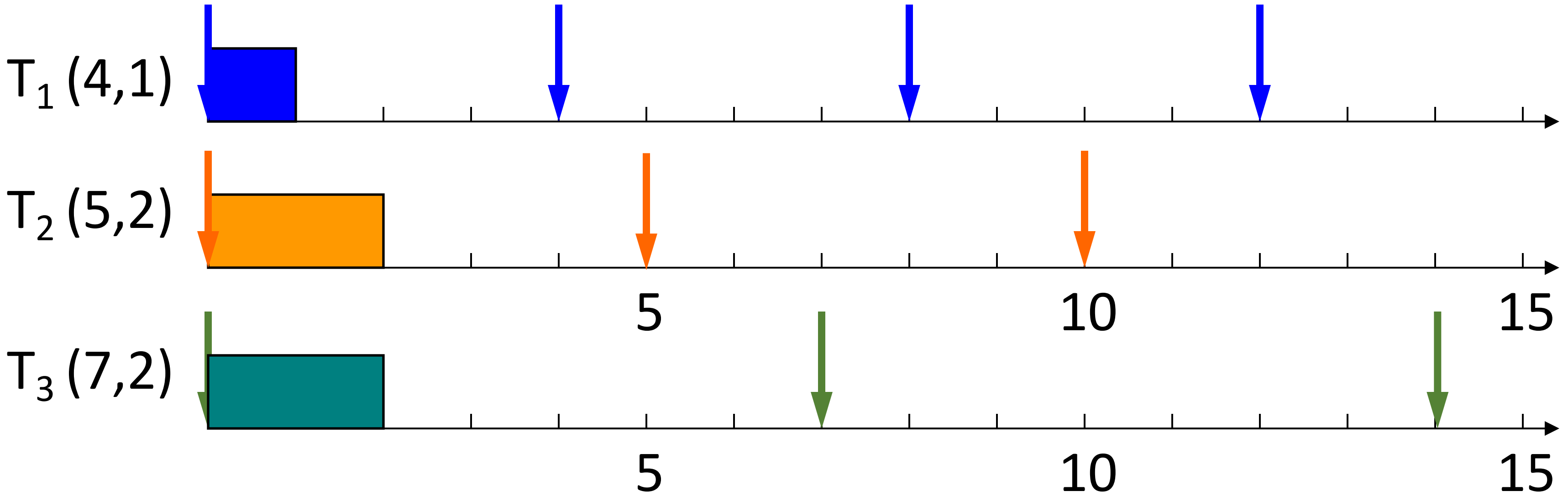
- Example:

T1
(4,1)

T2
(5,2)

T3
(7,2)

The Challenge



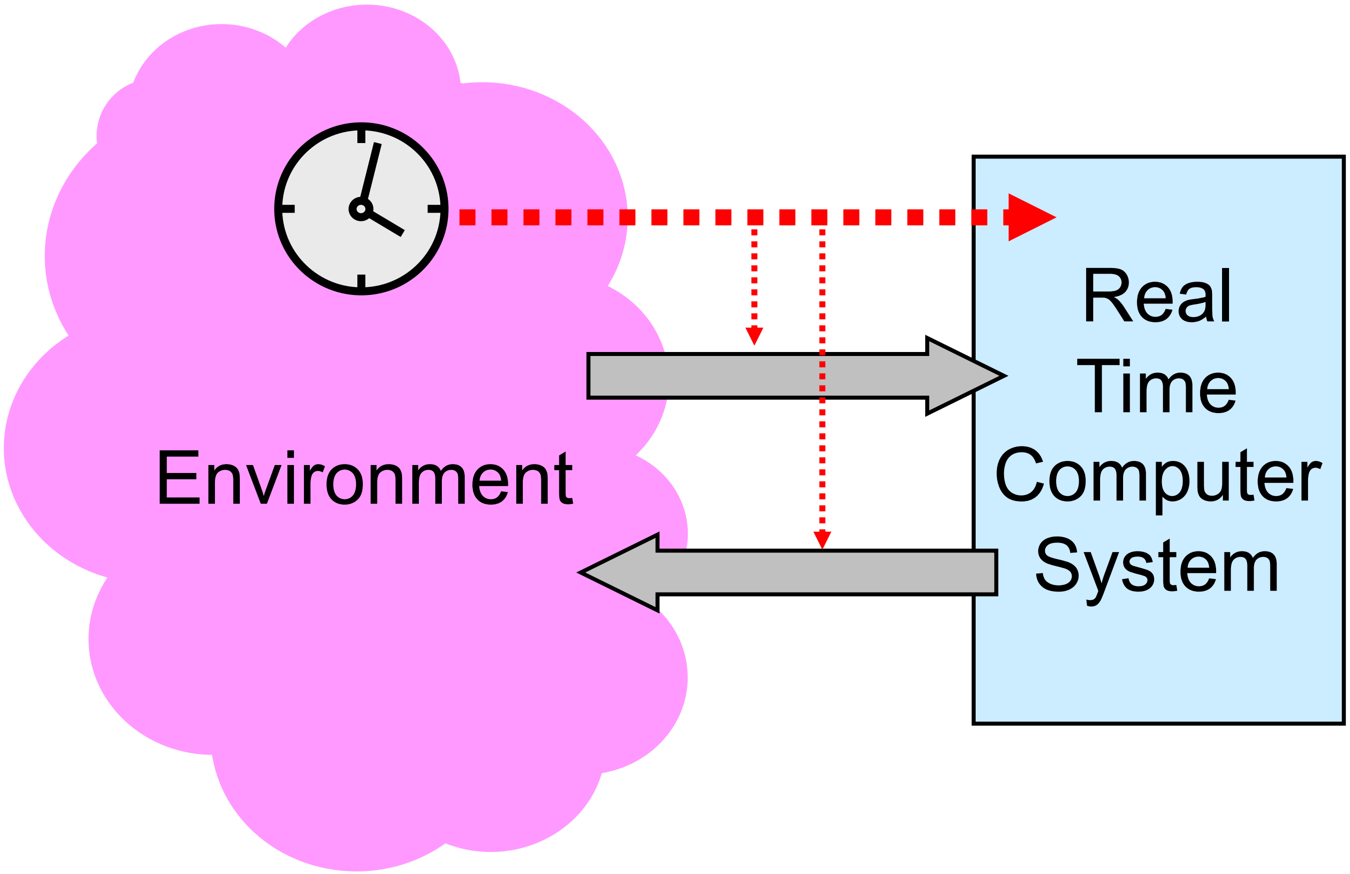
Real-Time System

*"A real-time computer system is a computer system where the **correctness** of the system behavior depends not only on the **logical results** of the computations, but also on the **physical time** when these results are produced."*

[Kopetz, RTS-Book 2011, p.2]

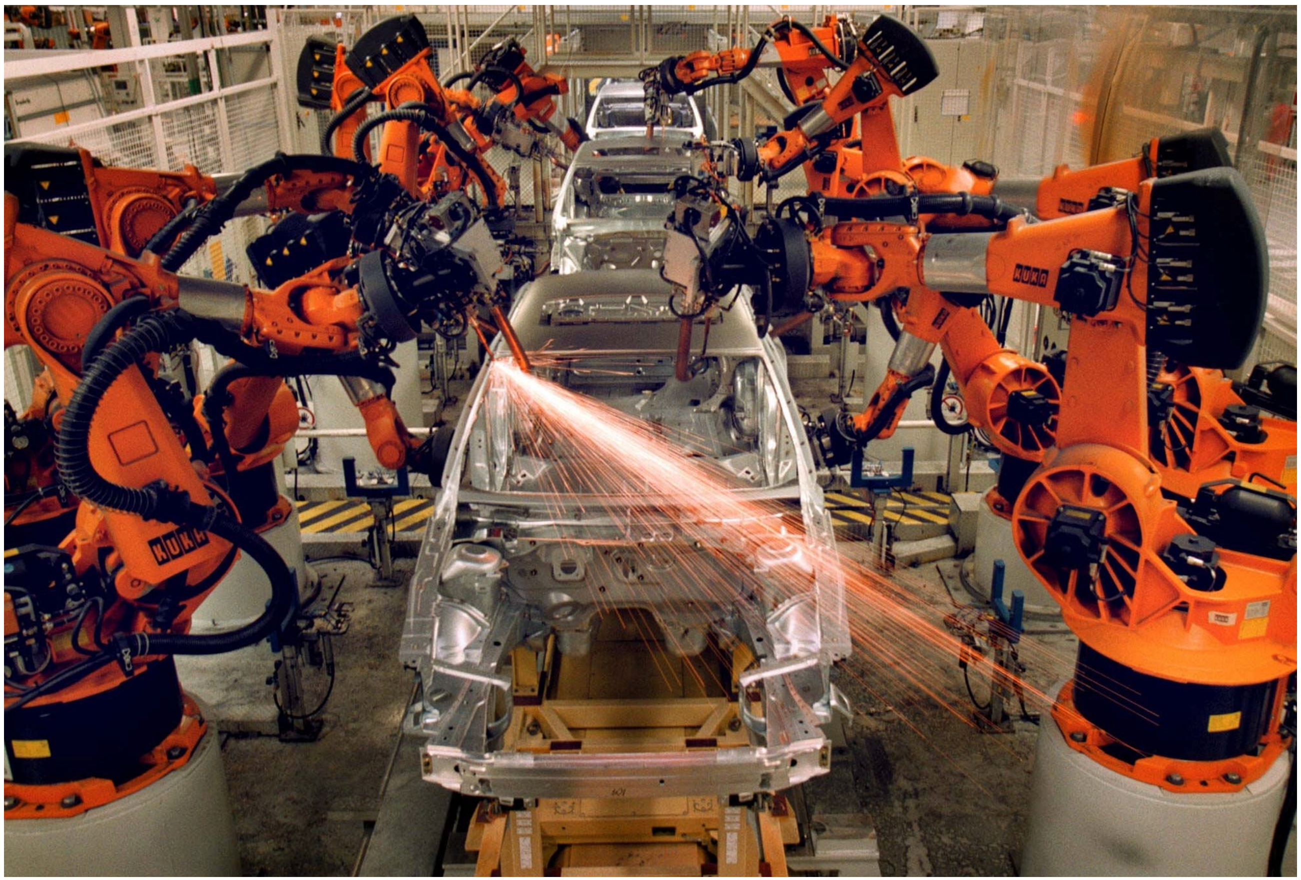
There is NO real application that does not have certain expectations about the timing of a computer system!

Real-Time System



Real-Time System

Example of Real-time Requirements

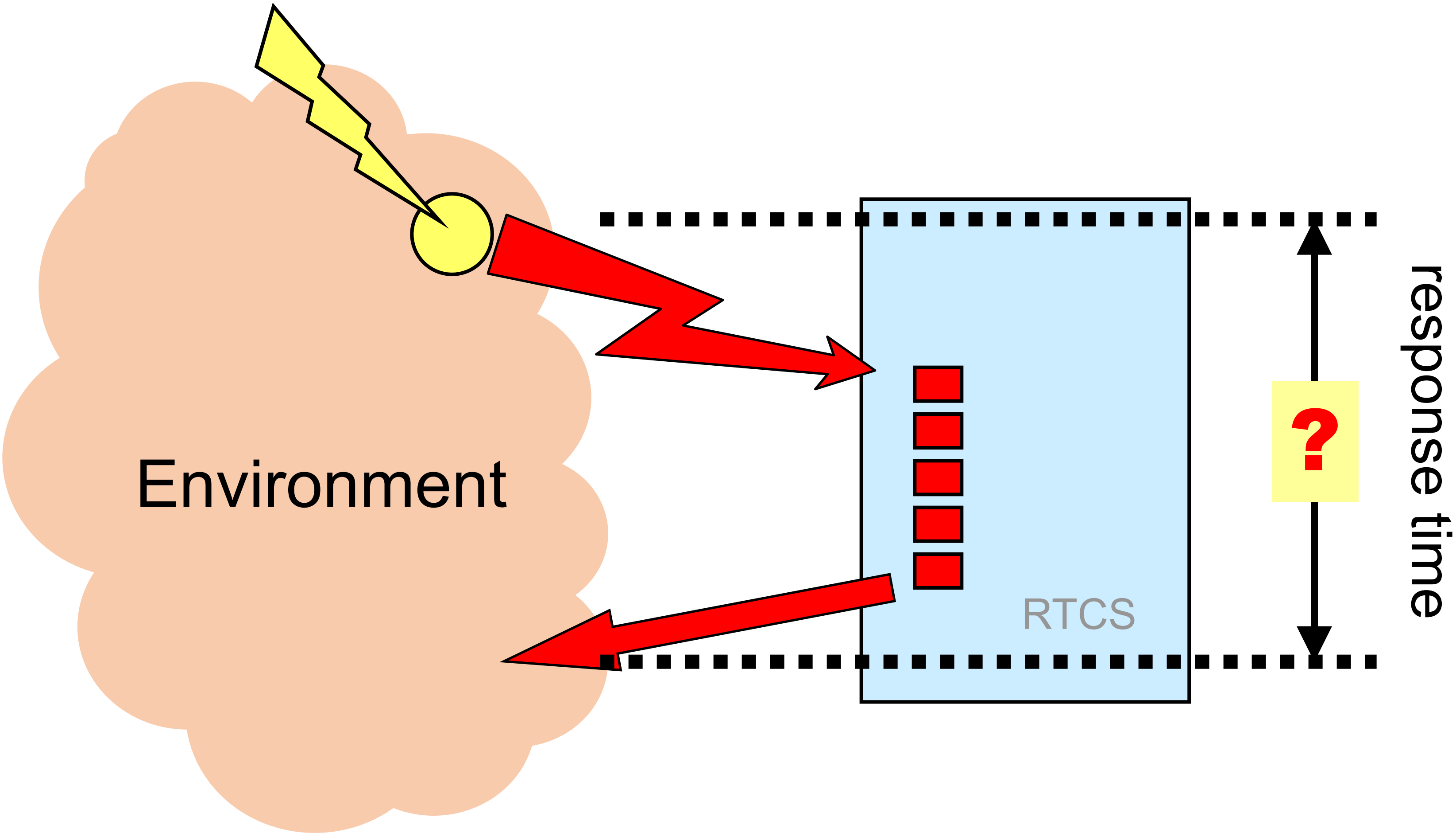


- Robot arms in car manfactory
- Multiple robot arms in use simultaneously
- Not only moving too slow, but also moving too fast can be problematic

Real-Time System

- Timing constraints on operations
  (e.g., deadline from event to completion of response)
- Hard real-time system:
  - Timing constraints must hold under all circumstances (even under peak load)
  - Failures may have severe consequences
    e.g., nuclear power station, medical equipment, airbag
- Note: real-time computing $\neq$ fast computing

How do we know the timing is right?



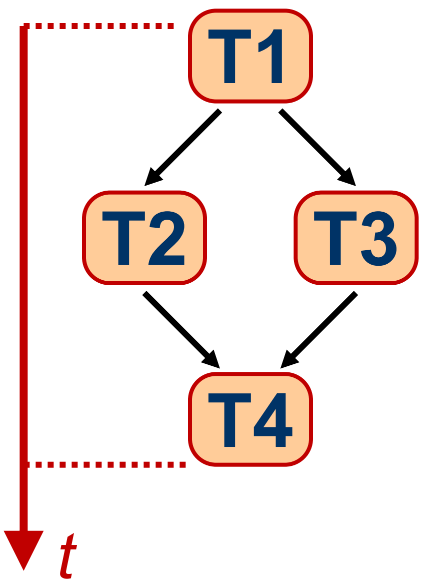
Time in RTS Construction

Design
Architecture, resource planning, schedules

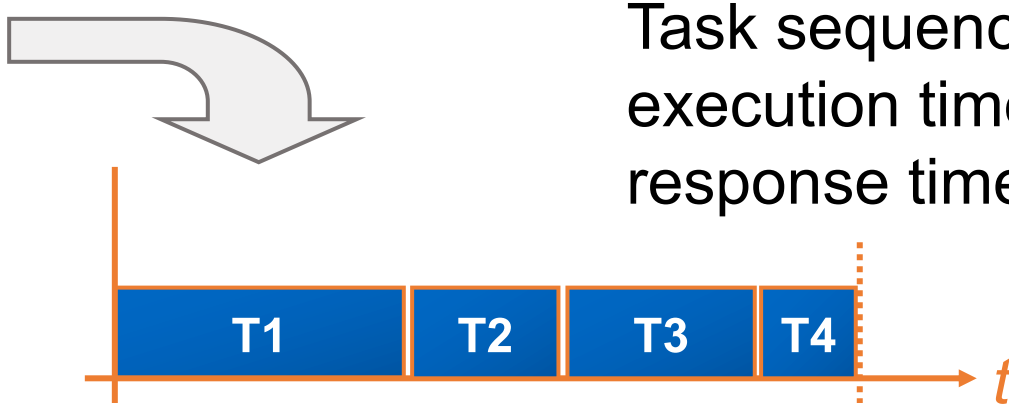
Implementation

Timing Analysis
Schedulability analysis, WCET analysis

From Design to Implementation



Task set with precedence constraints and deadline



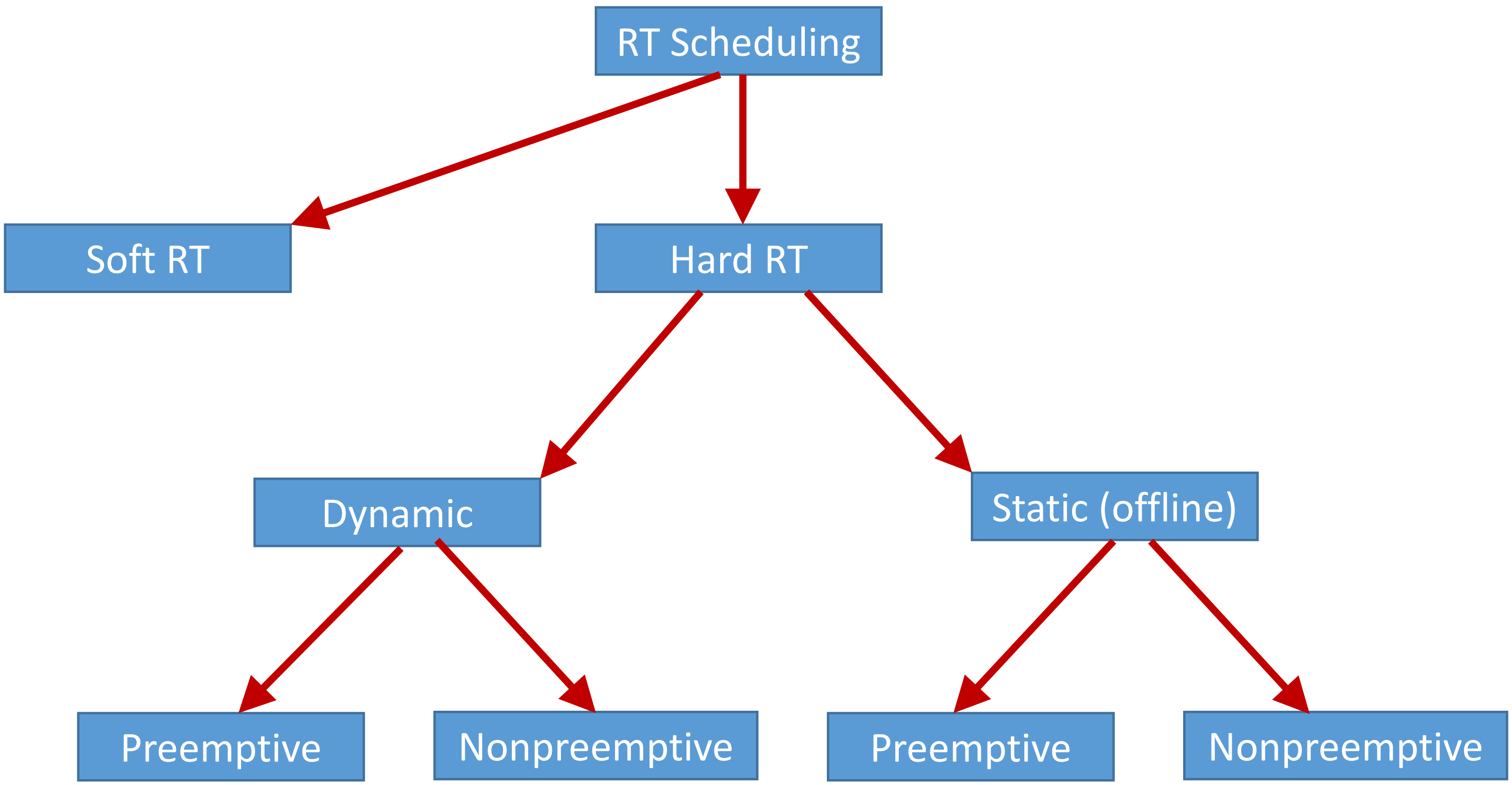
Task sequence: execution times, response time

Can we guarantee that: response time < deadline?

Real-Time Scheduling

- Application is subdivided into multiple tasks
- Instances of tasks are the schedulable objects, which are called **jobs**
- RT tasks have timing requirements → deadline
- Firm deadlines may not be missed.
- Dependent tasks have additional scheduling constraints:
    - precedence constraints (data flow)
    - resource constraints
      (additional shared resources besides CPU)

Taxonomy of Real-Time Scheduling



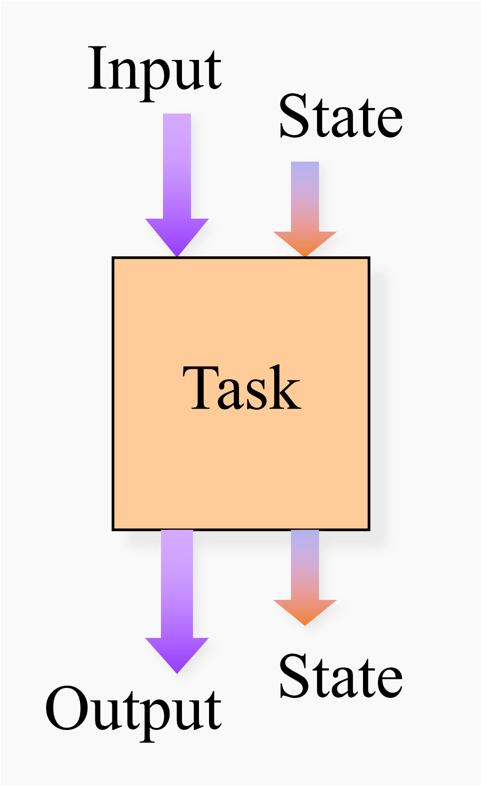
Simple Task

- Predictability of scheduling mechanisms is difficult if tasks use blocking communication and synchronisation mechanisms
- For many real-time scheduling methods only so-called simple tasks are allowed, to simplify their behavioural analysis

Simple Task

- Inputs available at start
- Outputs ready at the end
- No blocking inside
- No synchronization or communication inside
- Execution time variations only due to differences in
  - inputs
  - task state at start time

  (no external disturbances)



Static Scheduling

- Static scheduling is a scheduling method where scheduling decisions are resolved statically, i.e., offline before runtime
- Static scheduling assumes that for the application it is sufficient to fulfil jobs in an a-priory order
- In static scheduling tasks are in general executed periodically

Static Scheduling - Cyclic Executive

- The cyclic executive is a very simple method to schedule strictly periodic tasks
- The cyclic executive is basically a table of subroutine calls, each subroutine representing the implementation of a task
- The complete scheduling table is called the **major cycle**
  - the major cycle typically consists of a number of minor cycles, each of fixed duration
  - For example, four minor cycles of 25ms together would result in major cycles of 100ms
  - With major / minor cycles it is possible to realise multiple task periods

Static Scheduling - Cyclic Executive

- Example:
  schedule of 5 tasks with cyclic executive:

Task	Period
task_a	1
task_b	1
task_c	2
task_d	2
task_e	4

- In this example the cyclic executive requires four different minor cycles

```
# 1 major cycle, 4 minor cycles
loop (forever)
  wait_for_interrupt;
  exec_task(a);
  exec_task(b);
  exec_task(c);

  wait_for_interrupt;
  exec_task(a);
  exec_task(b);
  exec_task(d);
  exec_task(e);

  wait_for_interrupt;
  exec_task(a);
  exec_task(b);
  exec_task(c);

  wait_for_interrupt;
  exec_task(a);
  exec_task(b);
  exec_task(d);
end_loop
```

Cyclic Executive – Problems/Limitations

- The **major cycle** can become very long, i.e., including a lot of minor cycles in case of
  a) If task periods are **relative prime** to each other:
  e.g., given 3 tasks with these periods: p_1=5, p_2=7, p_3=13,
  then the major cycle is 5*7*13 = 455 minor cycles
  b) If task periods have a **big differences**:
  e.g., given 2 tasks with these periods: p_1=500, p_2=1, p_3=20,
  then the major cycle is 500 minor cycles
- Need to code a lot of almost similar minor cycles, which is tedious and error prone

Cyclic Executive – Problems/Limitations

- The concept of dividing the major cycle into equal long minor cycles has the additional problem that the developer has to decide in which minor cycle to put task calls so that the minor cycles are balanced.
- If one task has a **long execution-time**, it might be the case that it does not even fit into one minor cycle (since Cyclic Executive does not support preemption, the only way around this would be to split this task manually into smaller tasks that fit into individual minor cycles, which is very tedious)
- Thus, for above cases, more advanced scheduling methods are desirable, e.g., Rate-Monotonic Scheduling (RMS)

Rate Monotonic Scheduling (RMS)

- A static scheduling method that is more flexible than cyclic executive with respect of supporting multiple periods is **Rate Monotonic Scheduling** (**RMS**)

Rate Monotonic Scheduling (RMS) - Assumptions

A1: Tasks $\{T_i\}$ are periodic.
Period P_i = interval between two consecutive activations of task

A2: The execution time C_i of a task T_i is constant.

A3: The relative deadline D_i of a task T_i is equal to the period: $D_i = P_i$

A4: All tasks are independent
(i.e., no precedence constraints and no resource constraints)

Further assumptions:

A5: Tasks are preemptable, but no task can suspend itself

A7: All tasks are released as soon as they arrive

A8: Scheduling overhead is zero

Rate Monotonic Scheduling (RMS)

- Under the given assumptions, RMS is an **optimal static-priority scheduling** protocol for **single processor** systems

- Being optimal means that any other scheduling policy for above class of scheduling problems cannot perform better than RMS

RMS Protocol

- RMS assigns priority according to period
- A task with a shorter period has a higher priority
- Executes a job with the shortest period
- Task are characterised by $T_i(p,c)$, where
  - p is the period of task T_i
  - c is the execution time of task T_i (assumed to be constant)

RMS Protocol

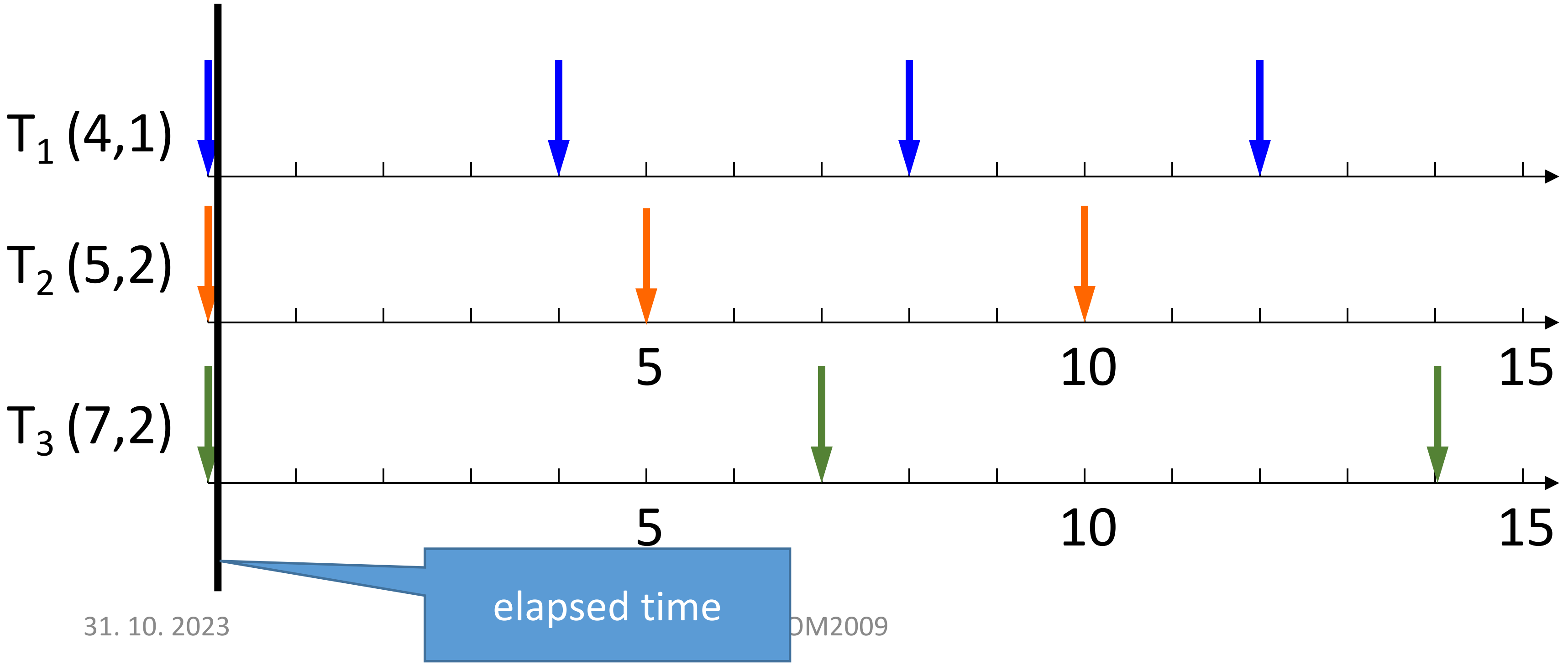
- Liu and Layland have shown in 1973 that RMS is schedulable if the utilisation factor U is given by

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

  where
  - n is the total number of tasks,
  - c_i the execution time, and
  - p_i the period of task T_i
- for large *n*, *U* approaches *ln2*, i.e., about 0.7

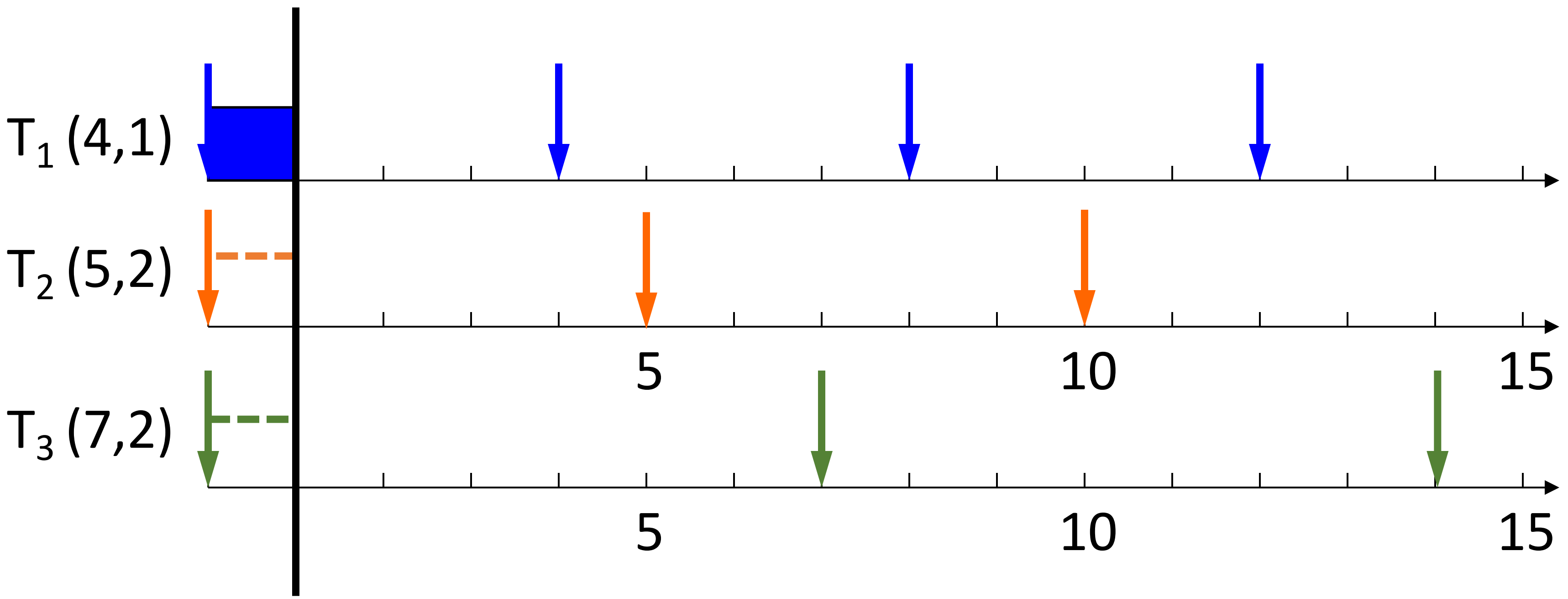
RMS Protocol Example

- Three tasks are assumed to arrive at the same time, which is the worst-case scenario
(arrows mark the arrival of a new task instance)



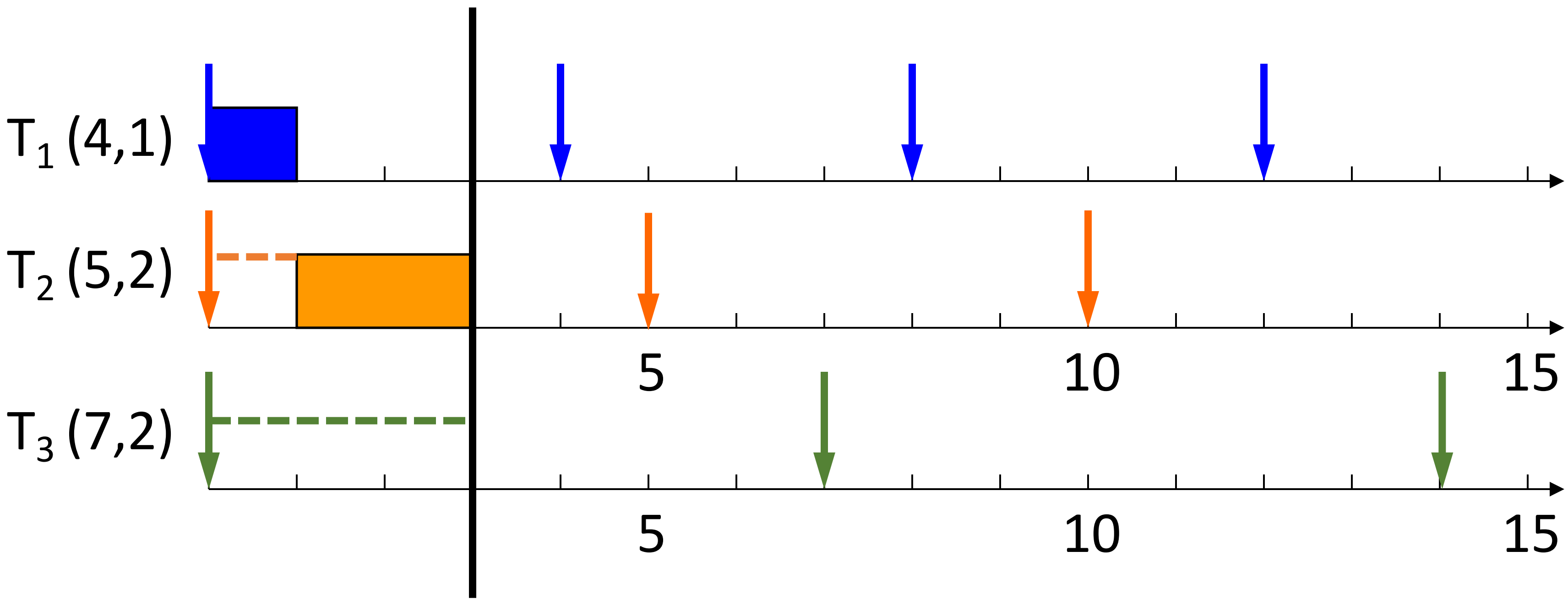
RMS Protocol

- First, T_1 gets executed, since it has the highest priority, i.e., shortest period
- T_2 and T_3 have to wait



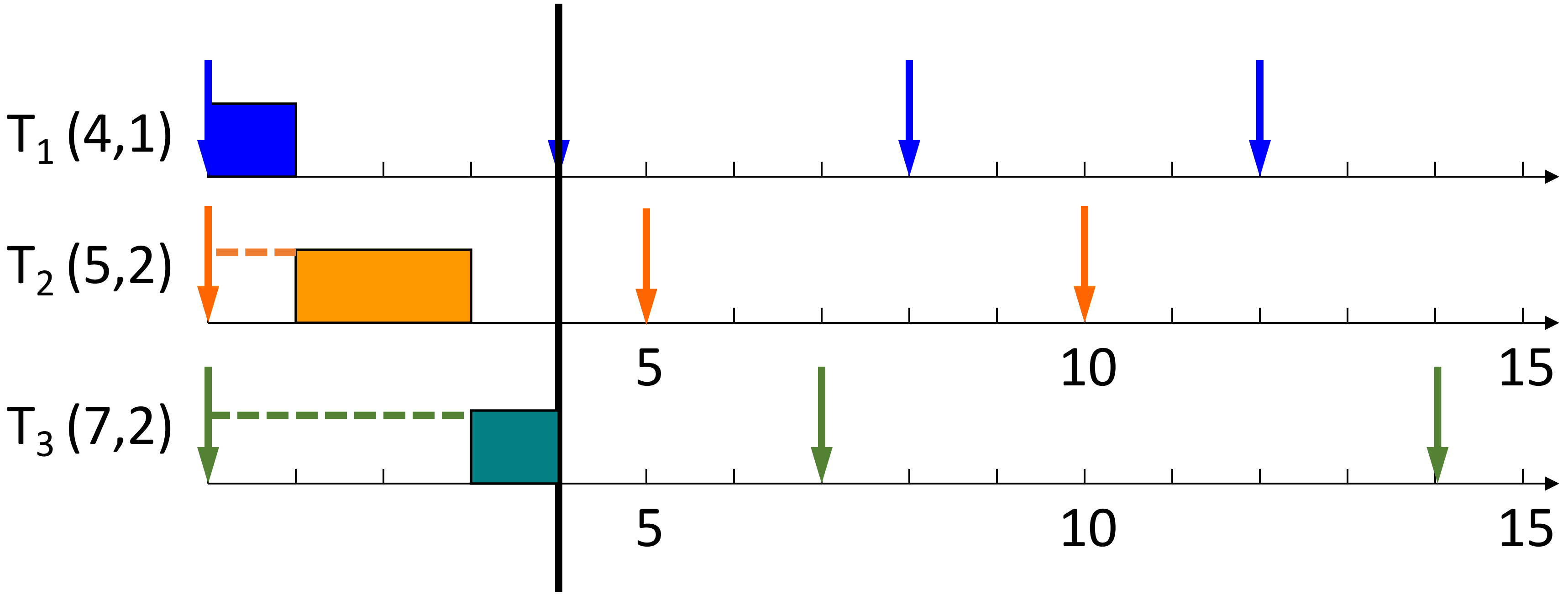
RMS Protocol

- Second, T_2 gets executed, since it has the highest priority after T_1



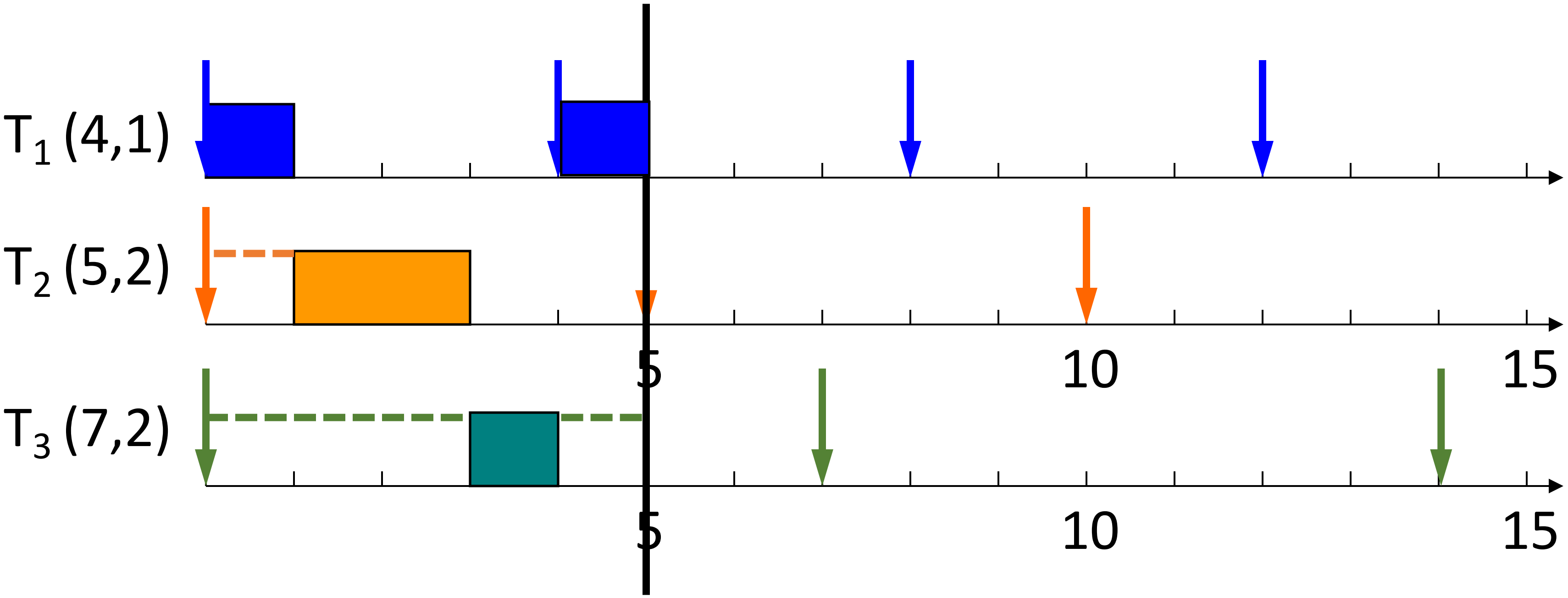
RMS Protocol

- At time t=3, T_3 starts execution, and gets preempted at t=4, due to the arrival of a new instance of task T_1



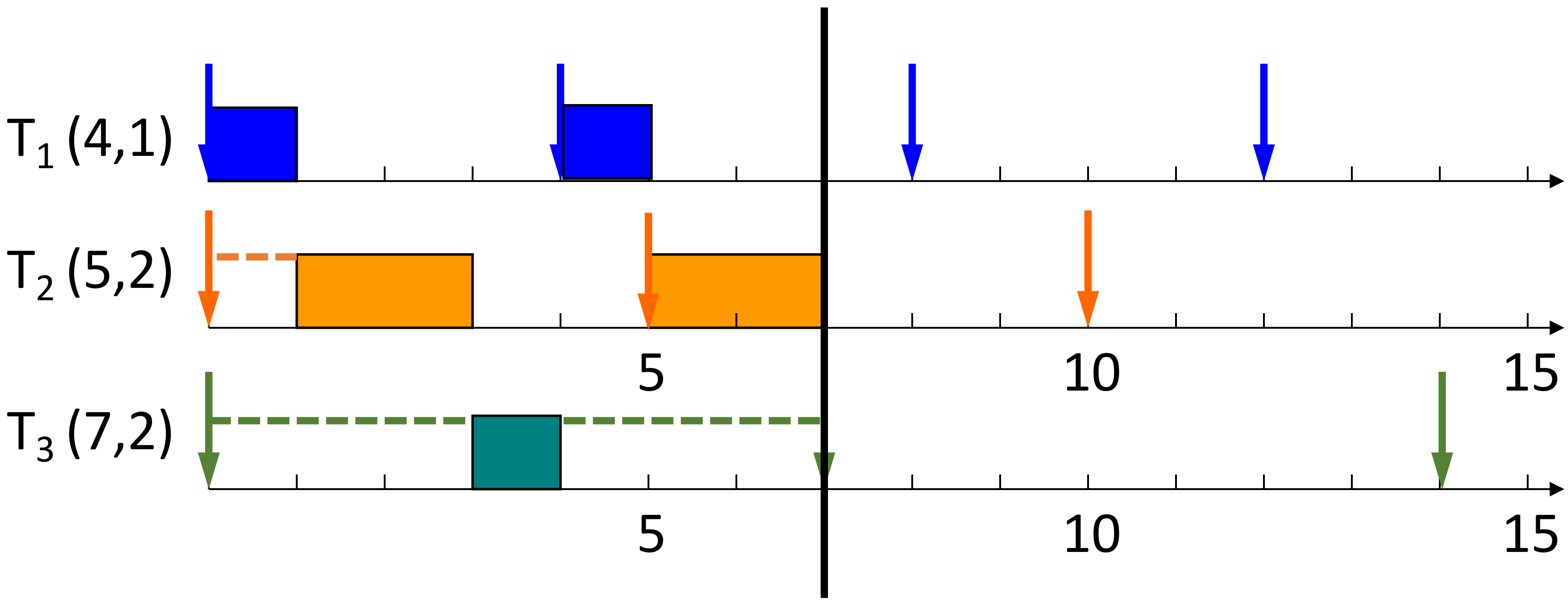
RMS Protocol

- At time t=4, T_1 starts execution, and finishes at t=5



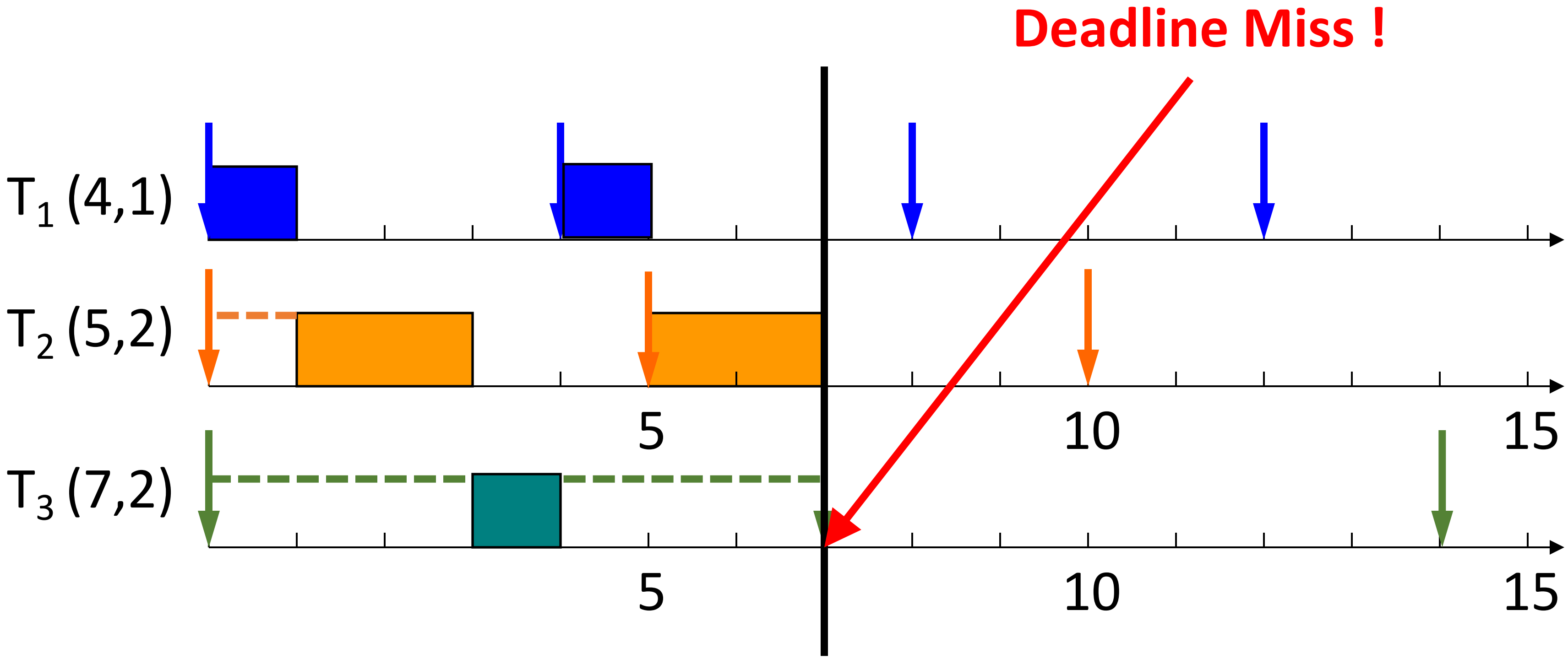
RMS Protocol

- At time t=5, a new instance of T_2 arrives and starts execution, and finishes at t=7



RMS Protocol

- At time t=7, the first instance of T_3 would be ready to be further executed.
- However, t=7 is also the deadline of T_3, which means T_3 misses its deadline

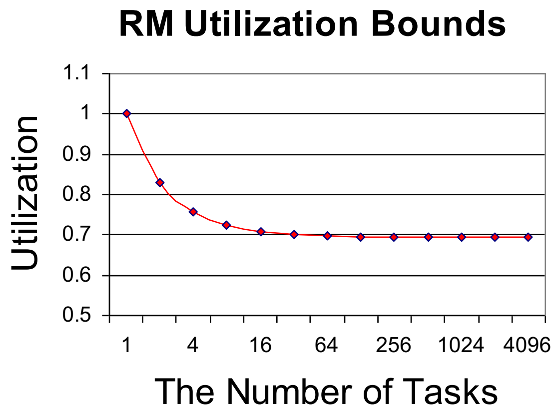


RMS – Utilization Bound

- Real-time system is schedulable under RM (sufficient condition) if

$$\sum U_i \leq n\,(2^{1/n}-1)$$

Term $n\,(2^{1/n}-1)$ approaches ln 2, i.e., about 0.7, for lim n→infinity



Liu & Layland, *"Scheduling algorithms for multi-programming in a hard-real-time environment"*, Journal of ACM, 1973.

RMS – Utilization Bound

- Real-time system is schedulable under RMS if

$$\sum U_i \leq n\,(2^{1/n}-1)$$

- Example: $T_1(4,1)$, $T_2(5,1)$, $T_3(10,1)$,

$$\sum U_i = 1/4 + 1/5 + 1/10 = 0.55$$
$$n\,(2^{1/n}-1)\,|_{n=3} = 3\,(2^{1/3}-1) \approx 0.78$$

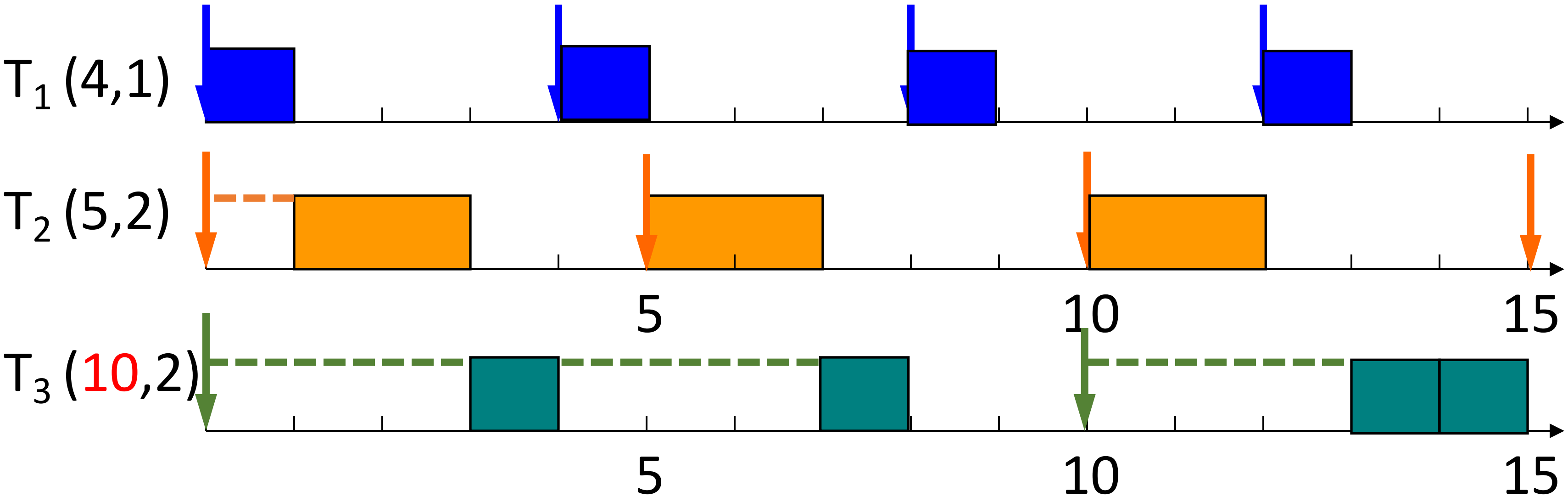
Thus, $\{T_1, T_2, T_3\}$ is schedulable under RMS.

Response Time Analysis of RMS

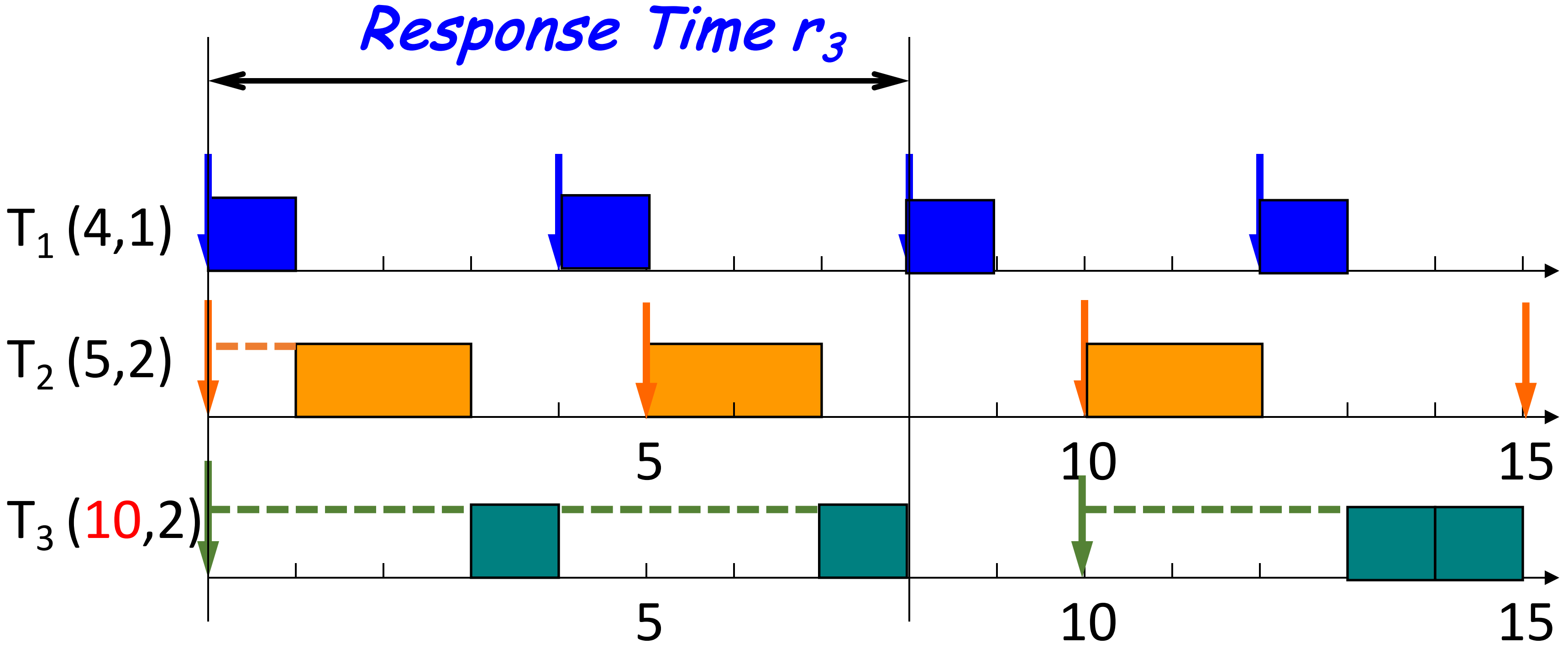
- Response time r_i
  ... duration from released time to finish time of task T_i

RMS - Response Time Analysis

- New example (this one is schedulable with RMS):



RMS - Response Time Analysis



RMS - Response Time Analysis

- r_x … response time of task T_x
- c_x … execution time of task T_x
- p_x … period of task T_x
- HP(T_x) … the set of all tasks with higher-priority than T_x

in our example
T1(4,1), T2(5,2), T3(10,2)
we have

- HP(T_1) = {}
- HP(T_2) = {T_1}
- HP(T_3) = {T_1, T_2},

Formula to calculate RMS response time:

$$\boxed{r_i = c_i + \sum_{T_k \in HP(T_i)} \left\lceil \frac{r_i}{p_k} \right\rceil \cdot c_k}$$

RMS - Response Time Analysis

- Real-time system is schedulable under RMS

if and only if $\boldsymbol{r_i \leq p_i}$ for all tasks $T_i(p_i, c_i)$

Joseph & Pandya, "*Finding response times in a real-time system*", The Computer Journal, 1986.

RMS - Response Time Analysis

Assumed periodic task set:

Task	***Period (p)***	***ExecTime (c)***
T1	4	1
T2	5	2
T3	10	2

Iterative fixed-point calculation of maximum response time per task:

r1	***r2***	***r3***
1	2	2
1	3	5
	3	6
		8
		8

$$\boxed{r_i = c_i + \sum_{T_k \in HP(T_i)} \left\lceil \frac{r_i}{p_k} \right\rceil \cdot c_k}$$

Discussion of RMS

- RMS is **optimal** among all fixed priority scheduling algorithms for scheduling periodic tasks where the deadlines of the tasks equal their periods
  - Scheduling protocols with dynamic priorities can handle higher utilisation, e.g., EDF
- Worst case load for RMS is the **critical instant** of all tasks released at the same time
- In case of **shared resources** (dependent tasks) RMS will be subject to priority inversion
  - Can be fixed by introducing **priority inheritance protocol**
  - But priority inheritance protocol does not prevent **deadlocks**

Outlook

- Next tutorial on programming a line follower

- Next lecture:
  Real-Time Scheduling II