

Aperiodic Task Scheduling

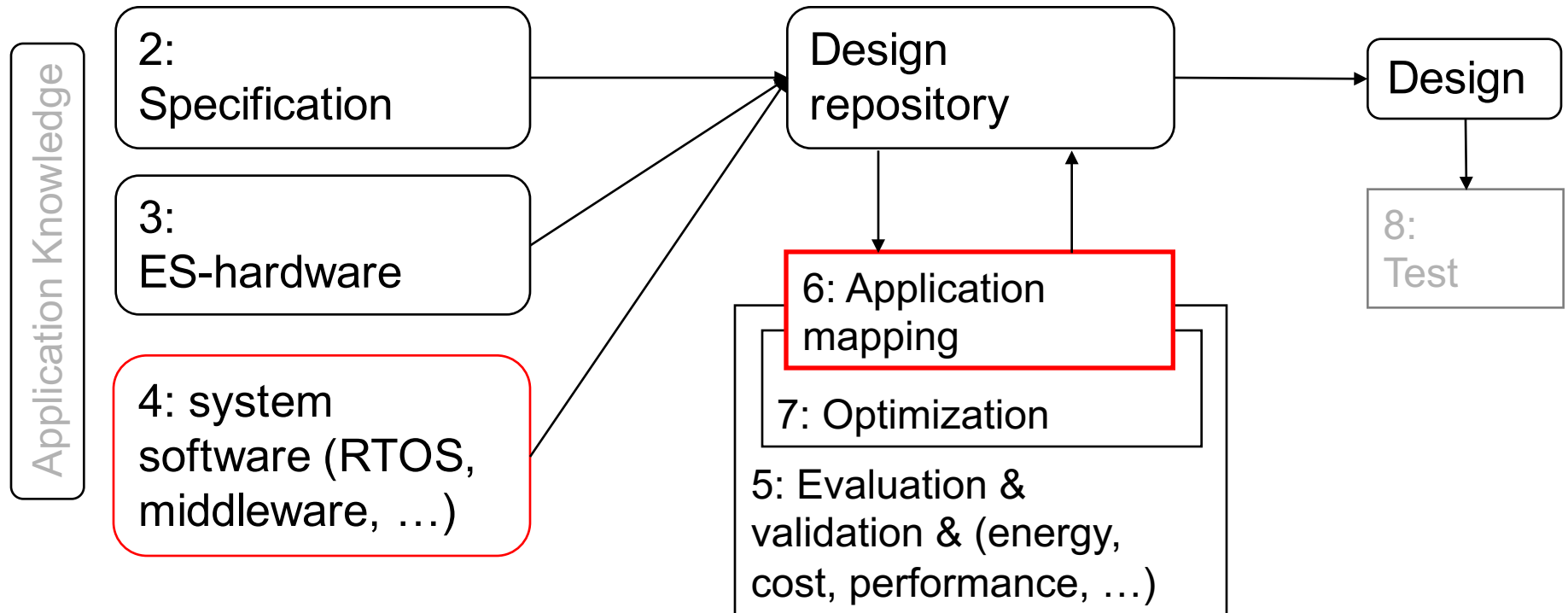
Jian-Jia Chen

(slides are based on
Peter Marwedel)

TU Dortmund, Informatik 12
Germany

2020年 01 月 07日

Structure of this course



Numbers denote sequence of chapters

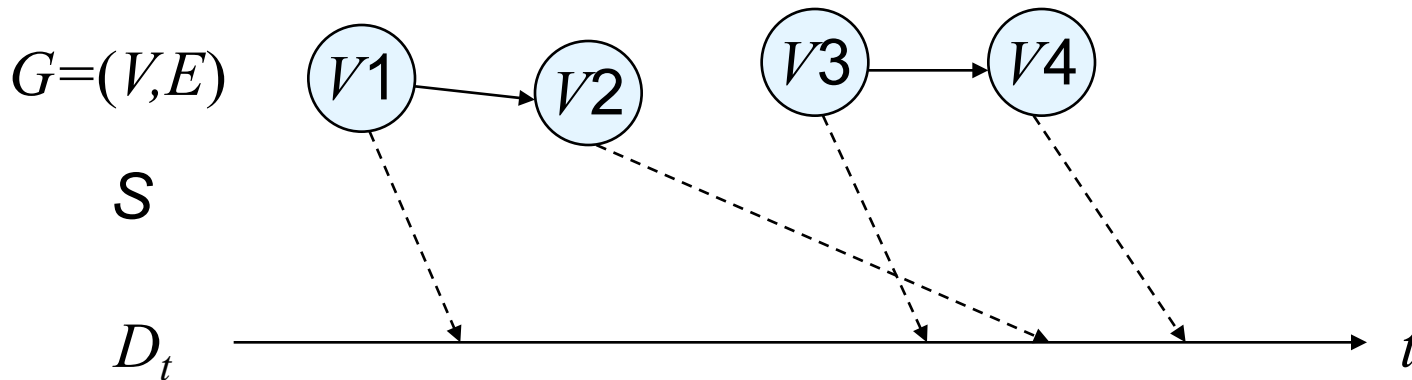
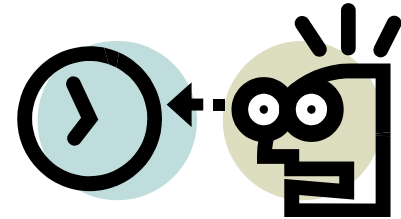
Real-time scheduling

Assume that we are given a task graph $G=(V,E)$.

Def.: A **schedule** S of G is a mapping

$$V \rightarrow D_t$$

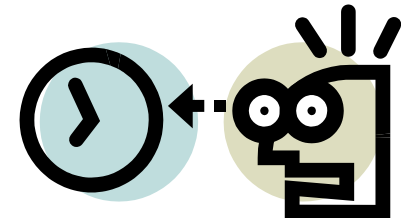
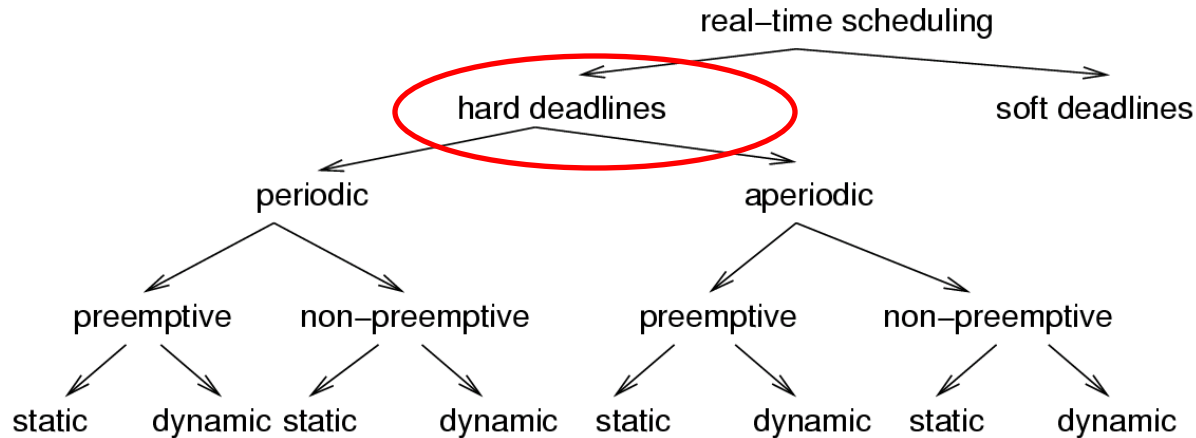
of a set of tasks V to start times from domain D_t .



Typically, schedules have to respect a number of constraints, incl. resource constraints, dependency constraints, deadlines.

Scheduling = finding such a mapping.

Hard and soft deadlines

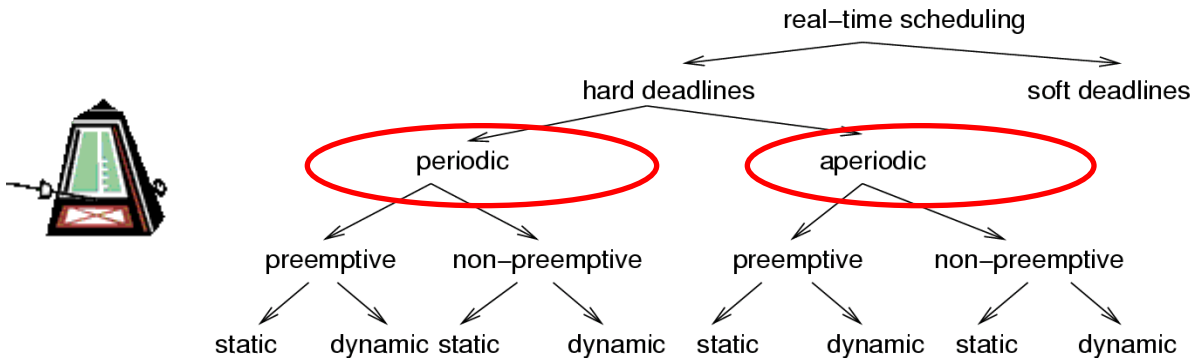


Def.: A time-constraint (deadline) is called **hard** if not meeting that constraint could result in a catastrophe [Kopetz, 1997].

All other time constraints are called **soft**.

We will focus on hard deadlines.

Periodic and aperiodic tasks

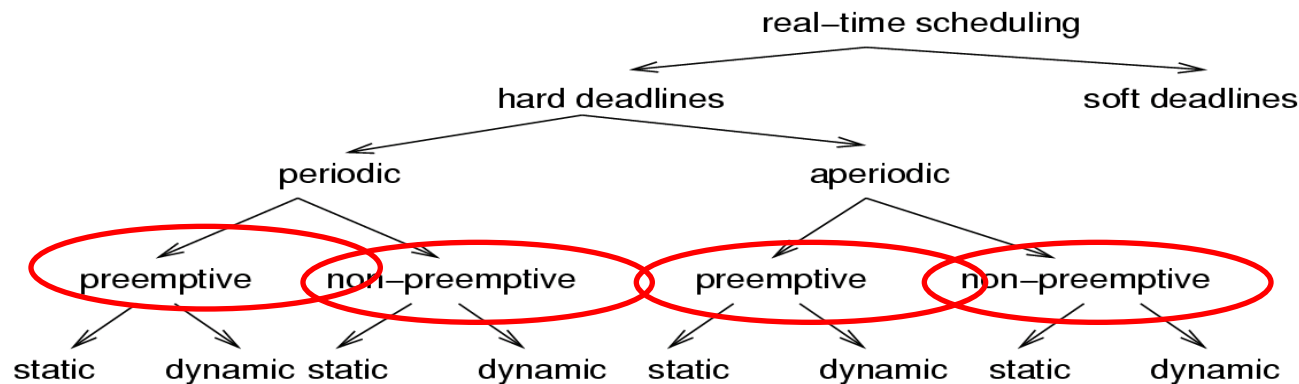


Def.: Tasks which must be executed once every T units of time are called **periodic** tasks. T is called their period. Each execution of a periodic task is called a **job**.

Def.: Tasks requesting the processor at unpredictable times are called **sporadic**, if there is a minimum separation between the times at which they request the processor.

All other tasks are called **aperiodic**.

Preemptive and non-preemptive scheduling



- **Non-preemptive schedulers:**

Jobs are executed until they are done.

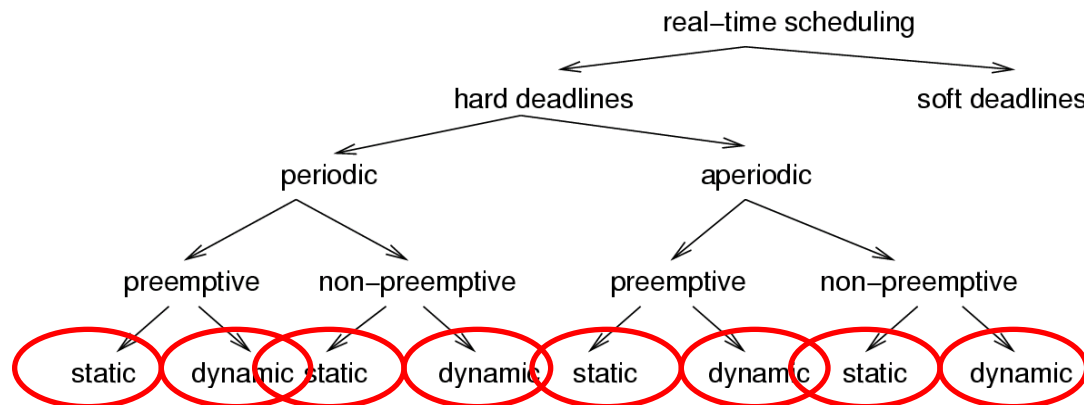
Response time for external events may be quite long.

- **Preemptive schedulers:** To be used if

- some tasks have long execution times or
- if the response time for external events to be short.

Dynamic/online scheduling

- **Dynamic/online scheduling:** Processor allocation decisions (scheduling) at run-time; based on the information about the tasks arrived so far.



Static/offline scheduling

- **Static/offline scheduling:**
Scheduling taking a priori knowledge about arrival times, execution times, and deadlines into account. Dispatcher allocates processor when interrupted by timer. Timer controlled by a table generated at design time.

Time	Action	WCET
10	<i>Start Task₁</i>	12
17	send M5	
22	<i>Start Task₃</i>	
38	<i>Start Task₂</i>	20
47	send M3	

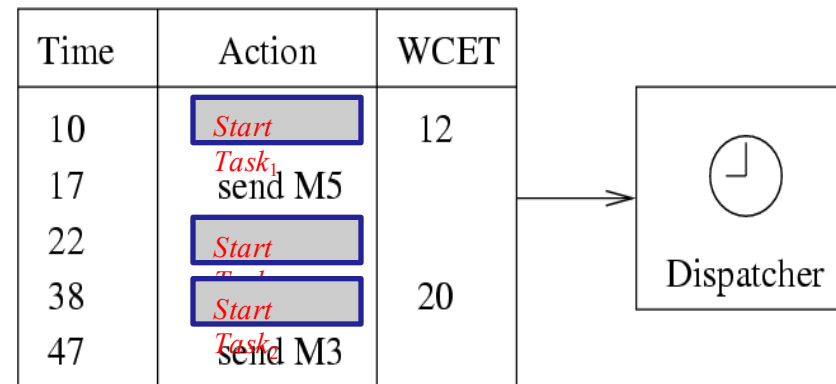
→ Dispatcher



Time-triggered systems (1)

*In an entirely time-triggered system, the temporal control structure of all tasks is established **a priori** by off-line support-tools. This temporal control structure is encoded in a **Task-Descriptor List (TDL)** that contains the cyclic schedule for all activities of the node. This schedule considers the required precedence and mutual exclusion relationships among the tasks such that an explicit coordination of the tasks by the operating system at run time is not necessary. ..*

The dispatcher is activated by the synchronized clock tick. It looks at the TDL, and then performs the action that has been planned for this instant [Kopetz].



Time-triggered systems (2)

... pre-run-time scheduling is often the only practical means of providing predictability in a complex system.
[Xu, Parnas].

It can be easily checked if timing constraints are met.
The disadvantage is that the response to sporadic events may be poor.

Centralized and distributed scheduling

- **Mono- and multi-processor scheduling:**
 - Simple scheduling algorithms handle single processors,
 - more complex algorithms handle multiple processors.
 - algorithms for homogeneous multi-processor systems
 - algorithms for heterogeneous multi-processor systems (includes HW accelerators as special case).
- **Centralized and distributed scheduling:**

Multiprocessor scheduling either locally on 1 or on several processors.

Possible Statements regarding Schedulability

- If A holds, then the task system is schedulable by an algorithm
- If the task system is schedulable by an algorithm, then B holds
- If and only if C holds, the task system is schedulable by an algorithm

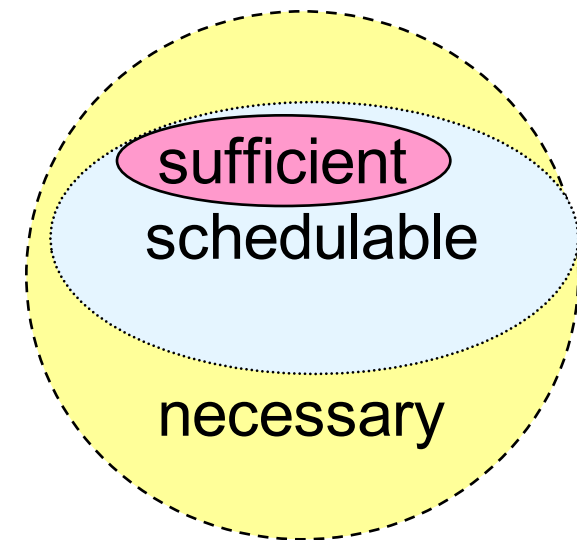
Schedulability

Set of tasks is **schedulable** under a set of constraints, if a schedule exists for that set of tasks & constraints.

Exact tests are NP-hard in many situations.

Sufficient tests: sufficient conditions for schedule checked.

Necessary tests: checking necessary conditions. Used to show no schedule exists. There may be cases in which no schedule exists & we cannot prove it.



Classical scheduling algorithms for aperiodic systems



These slides use Microsoft clip arts.
Microsoft copyright restrictions apply.

Aperiodic scheduling:

- Scheduling with no precedence constraints -

Let $\{\tau_i\}$ be a set of tasks. Let:

- c_i be the execution time of τ_i ,
- d_i be the **absolute deadline**
 - I will use deadline in this case when we have only aperiodic tasks
- r_i be the **arrive time**
- l_i be the **laxity** or **slack**, defined as $l_i = d_i - c_i - r_i$
- f_i be the finishing time.

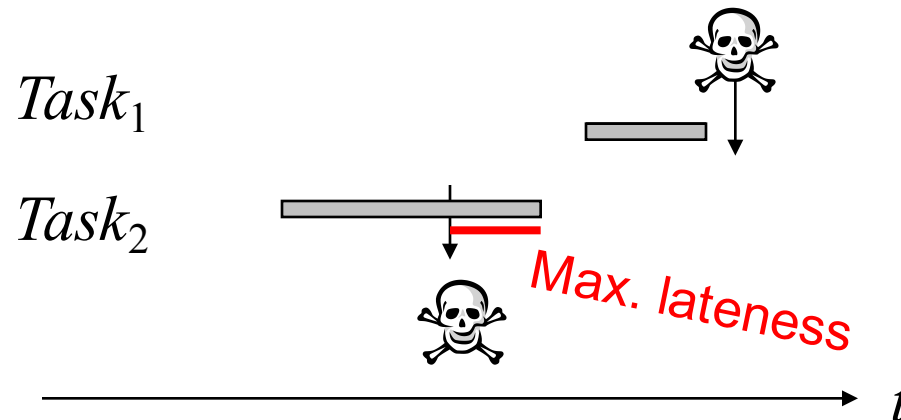
Cost functions

Cost function: Different algorithms aim at minimizing different functions.

Def.: Maximum lateness =

$\max_{\text{all tasks}} (\text{completion time} - \text{deadline})$

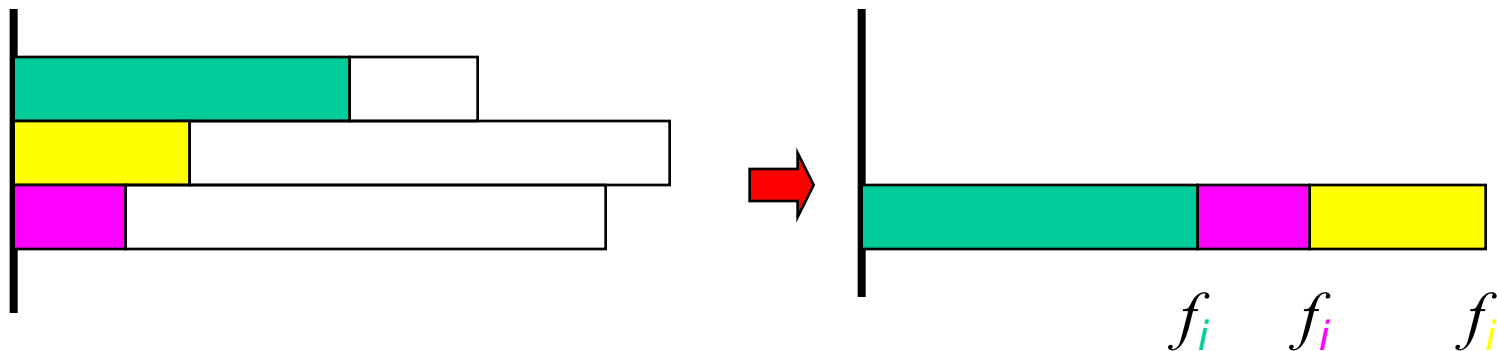
Is < 0 if all tasks complete before deadline.



Uniprocessor with equal arrival times

Preemption is useless.

Earliest Due Date (EDD): Execute task with earliest due date (deadline) first.



EDD requires all tasks to be sorted by their (absolute) deadlines. Hence, its complexity is $O(n \log(n))$.

Optimality of EDD

EDD is optimal for minimizing the maximum lateness:
Given a set of n independent tasks, any algorithm that executes the tasks in order of non-decreasing (absolute) deadlines is optimal with respect to minimizing the maximum lateness.

Proof (See Buttazzo, 2002):

- Let S be a schedule produced by any algorithm A
- If $S \neq$ the schedule of EDD $\rightarrow \exists \tau_a, \tau_b, d_a \leq d_b, \tau_b$ immediately precedes τ_a in S .
- Let S' be the schedule obtained by exchanging τ_a and τ_b .

Exchanging τ_a and τ_b cannot increase lateness

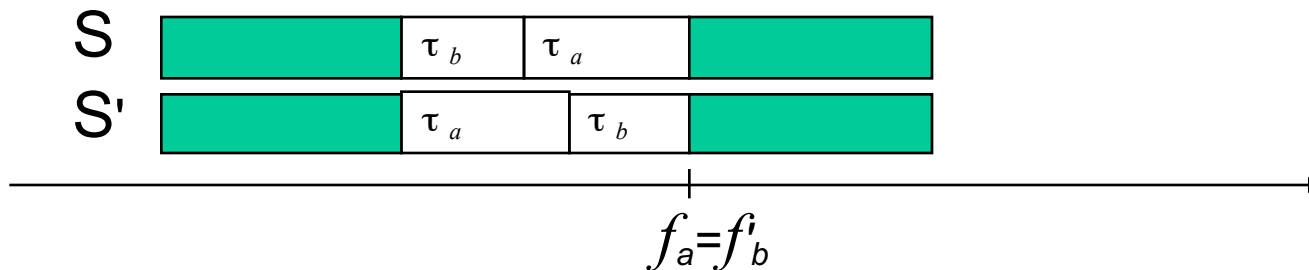
Max. lateness for τ_a and τ_b in S is $L_{max}(a,b) = f_a - d_a$

Max. lateness for τ_a and τ_b in S' is $L'_{max}(a,b) = \max(L'_a, L'_b)$

Two possible cases

1. $L'_a \geq L'_b$: $\rightarrow L'_{max}(a,b) = f'_a - d_a < f_a - d_a = L_{max}(a,b)$
since T_a starts earlier in schedule S' .
2. $L'_a < L'_b$: $\rightarrow L'_{max}(a,b) = f'_b - d_b = f_a - d_b \leq f_a - d_a = L_{max}(a,b)$
since $f_a = f'_b$ and $d_a \leq d_b$

$\Rightarrow L'_{max}(a,b) \leq L_{max}(a,b)$



EDD is optimal

- ➡ Any schedule S with lateness L can be transformed into an EDD schedule S^n with lateness $L^n \leq L$, which is the minimum lateness.
- ➡ EDD is optimal for minimizing the maximum lateness (q.e.d.)
- ➡ EDD is optimal for meeting the deadlines
- ➡ Note that EDD is not always optimal for other cost functions

Sufficient and Necessary Schedulability Tests for EDD

☞ A set of aperiodic tasks arriving at the same time (says, 0) is schedulable (by EDD) if and only if

$$\forall k, \quad \sum_{\tau_j: d_j \leq d_k} c_j \leq d_k$$

- Proof for if: this simply comes from the evaluation of EDD
- Proof for only if: this simply comes from the fact that the demand must be no more than the available time

Earliest Deadline First (EDF): - Horn's Theorem -

Different arrival times: Preemption potentially reduces lateness.

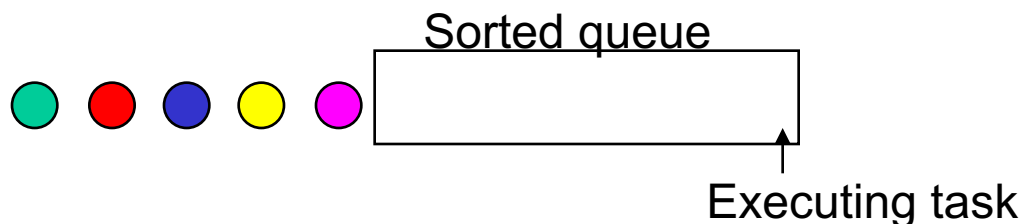
Theorem [Horn74]: Given a set of n independent tasks with arbitrary arrival times, any algorithm that at any instant executes the task with the earliest absolute deadline among all the ready tasks is optimal with respect to minimizing the maximum lateness.

Earliest Deadline First (EDF): - Algorithm -

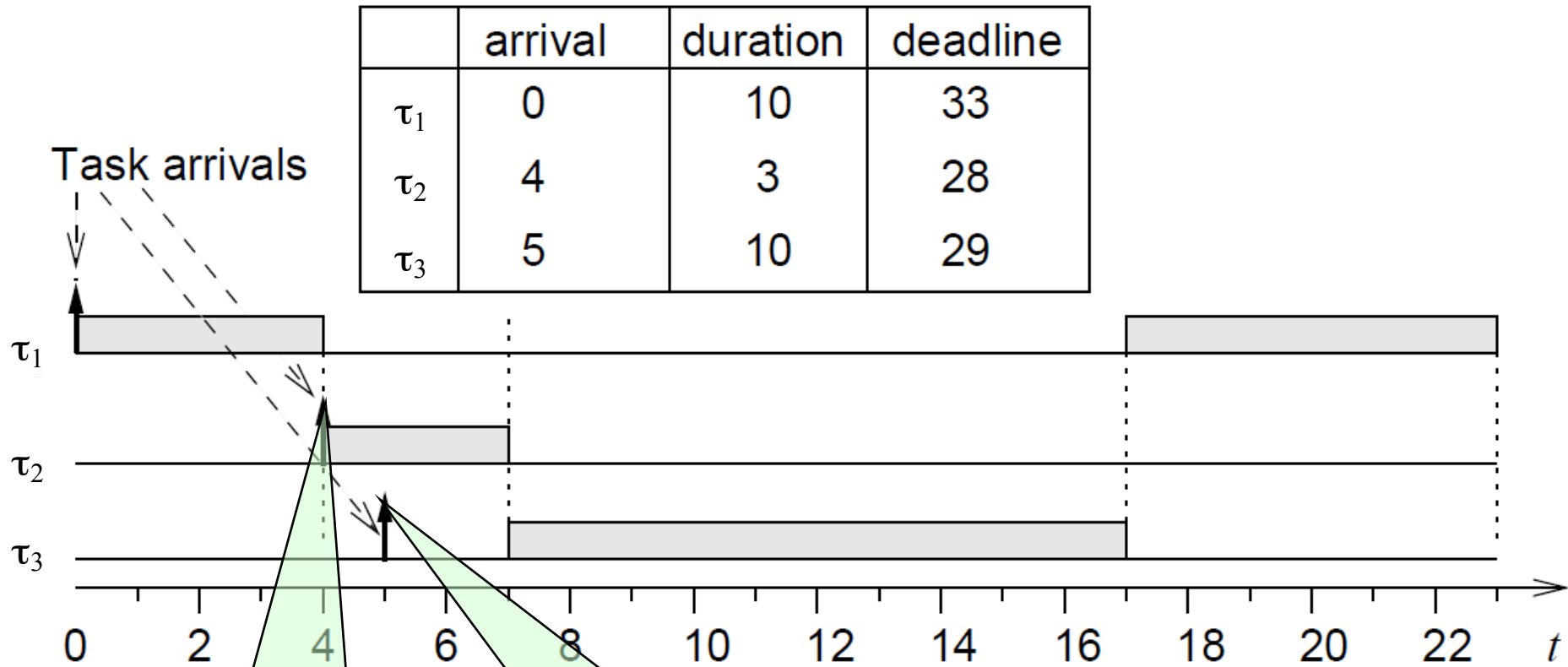
Earliest deadline first (EDF) algorithm:

- Each time a new ready task arrives:
- It is inserted into a queue of ready tasks, sorted by their **absolute** deadlines. Task at head of queue is executed.
- If a newly arrived task is inserted at the head of the queue, the currently executing task is preempted.

Straightforward approach with sorted lists (full comparison with existing tasks for each arriving task) requires run-time $O(n^2)$; (less with binary search or bucket arrays).



Earliest Deadline First (EDF): Example -



Earlier deadline
 ☞ preemption

Later deadline
 ☞ no preemption

Optimality of EDF

To be shown: EDF minimizes maximum lateness.

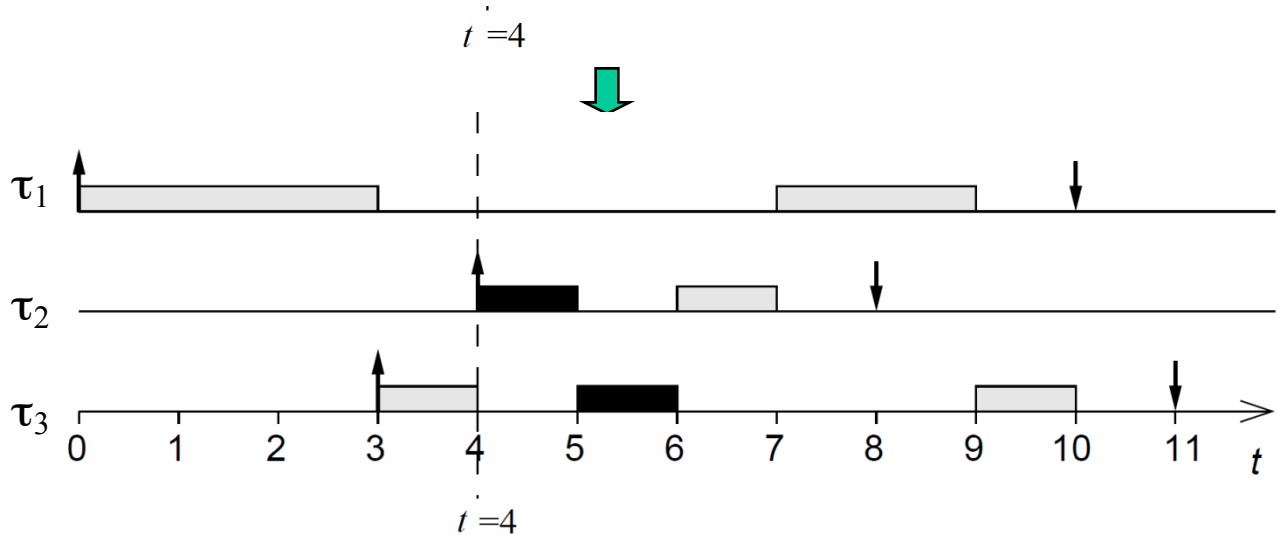
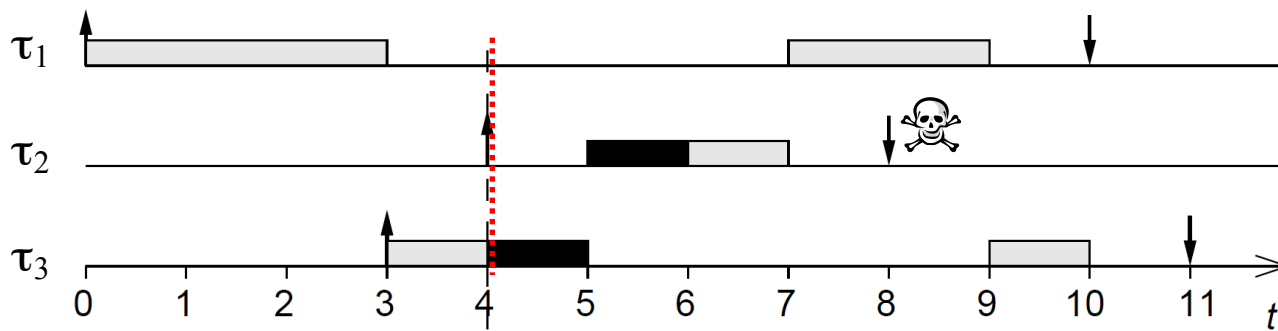
Proof (Buttazzo, 2002):

- Let S be a schedule produced by generic schedule A
- Let S_{EDF} : schedule produced by EDF
- Preemption allowed: tasks executed in disjoint time intervals
- S divided into time slices of 1 time unit each
- Time slices denoted by $[t, t+1)$
- Let $S(t)$: task executing in $[t, t+1)$
- Let $E(t)$: the unfinished task which, at time t , has the earliest absolute deadline
- Let $t_E(t)$: time ($\geq t$) at which the next slice of task $E(t)$ begins its execution in schedule S

Optimality of EDF (2)

If $S \neq S_{EDF}$, then there exists time $t: S(t) \neq E(t)$

Idea: swapping $S(t)$ and $E(t)$ cannot increase max. lateness.



Optimality of EDF (3)

Algorithm **interchange**:

```
{ for ( $t=0$  to  $D-1$ ) {  
  if ( $S(t) \neq E(t)$ ) {  
     $S(t_{E(t)}) = S(t)$ ;  
     $S(t) = E(t)$ ; }}}
```

Using the same argument as in the proof of EDD, it is easy to show that swapping cannot increase maximum lateness; hence EDF is optimal for minimizing the maximum lateness.

Does **interchange** preserve schedulability?

No, the argument is slightly buggy. Why not? How to fix it?
q.e.d.

Sufficient and Necessary Schedulability Tests for EDF

☞ A set of aperiodic tasks is schedulable (by EDD) if and only if

$$\forall r_i < d_k, \quad \sum_{\tau_j: r_i \leq r_j \text{ and } d_j \leq d_k} c_j \leq d_k - r_i$$

- Proof for only if (**necessary test**): this simply comes from the fact that the demand must be no more than the available time
- Proof for if (**sufficient test**)

Proof: Sufficient Schedulability Test for EDF

Proof by contrapositive:

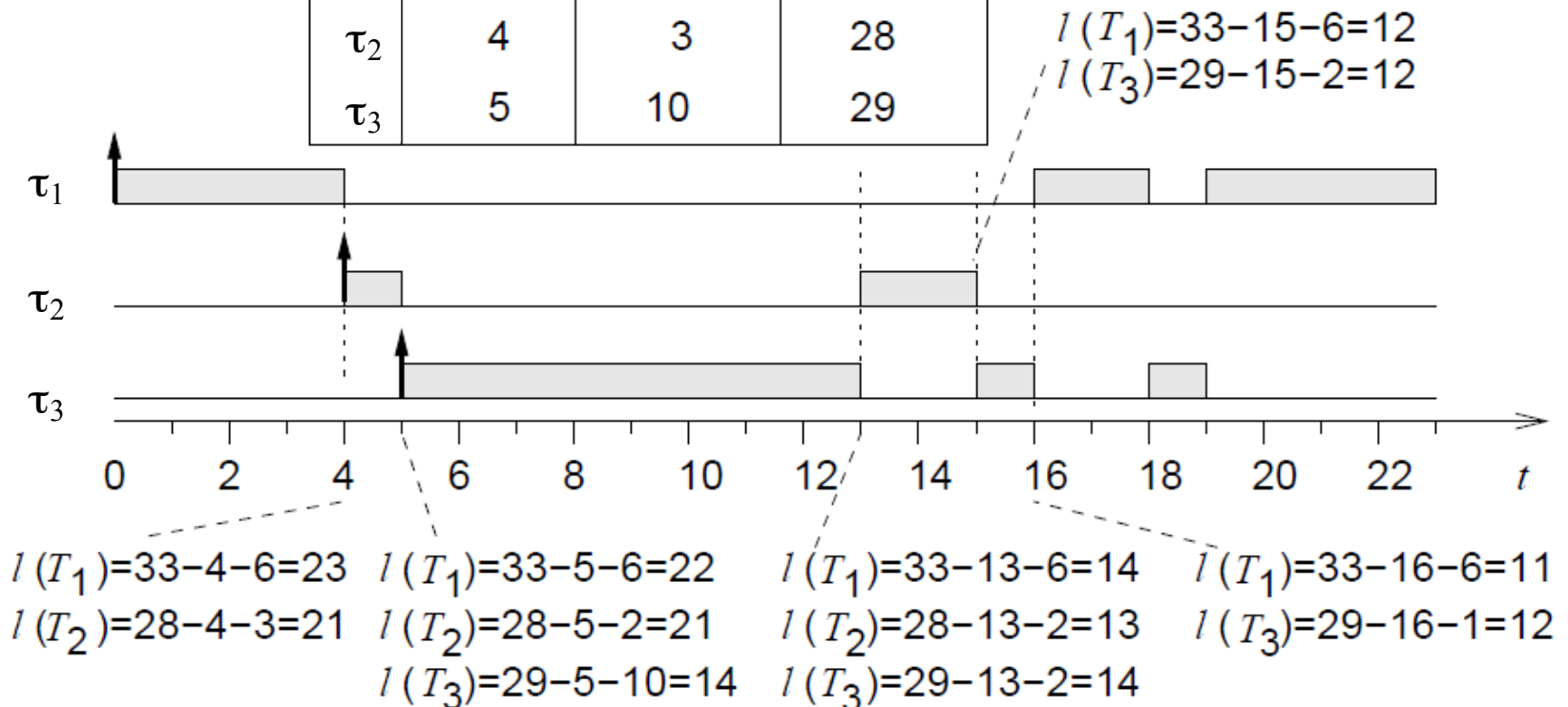
- Suppose that EDF schedule does not meet the deadline
- Let task τ_k be the first task which misses its absolute deadline d_k
- Let t_0 be the last instant before d_k , at which either the processor is idle or the processor executes a task with absolute deadline larger than d_k
- By EDF, t_0 must be an arrival time of a job, called τ_i
- Therefore, t_0 is equal to r_i
- Due to the deadline miss of τ_k , the processor must be busy between r_i and d_k
- The processor executes only the jobs arriving no earlier than r_i and with absolute deadline less than or equal to d_k
- Therefore, we conclude the proof by showing that

$$\exists r_i < d_k, \quad \sum_{\tau_j: r_i \leq r_j \text{ and } d_j \leq d_k} c_j > d_k - r_i$$


Least laxity (LL), Least Slack Time First (LST)

Priorities = decreasing function of the laxity
 (lower laxity \rightarrow higher priority); changing priority; preemptive.

	arrival	duration	deadline
τ_1	0	10	33
τ_2	4	3	28
τ_3	5	10	29



Properties

- Not sufficient to call scheduler & re-compute laxity just at task arrival times.
- Overhead for calls of the scheduler.
- Many context switches.
- **Detects missed deadlines early.**
- LL is also an optimal scheduling for mono-processor systems to meet the deadlines.
- Dynamic priorities  cannot be used with a fixed prio OS.
- LL scheduling requires the knowledge of the execution time.

Scheduling without preemption (1)

Lemma: If preemption is not allowed, optimal schedules may have to leave the processor idle at certain times.

Proof: Suppose: optimal schedulers never leave processor idle.

Scheduling without preemption (2)

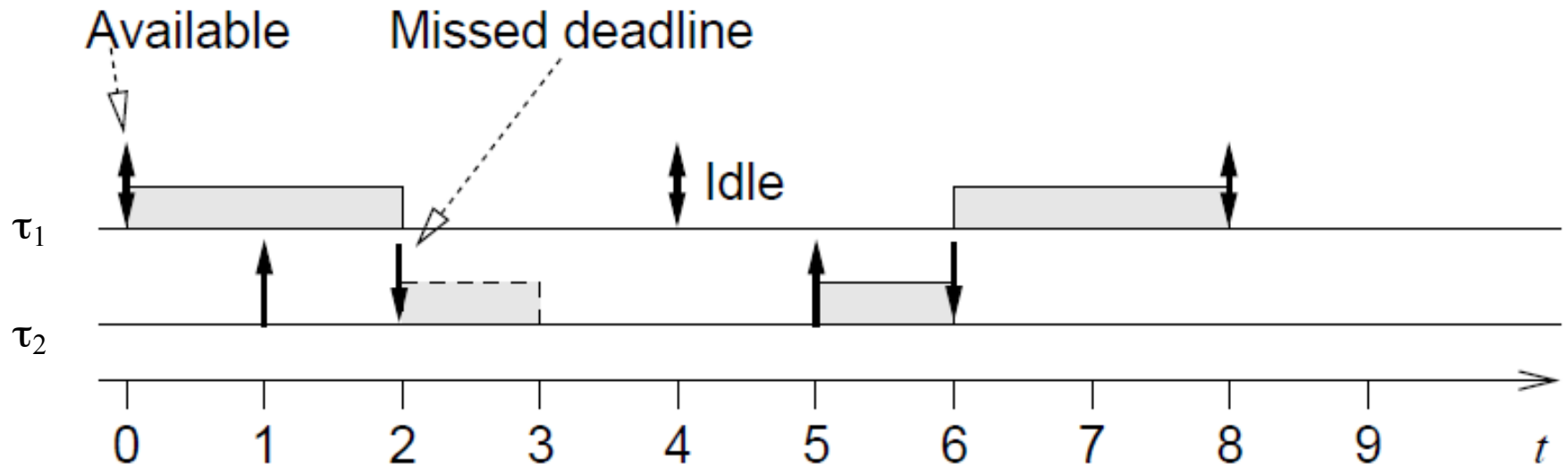
τ_1 : periodic, available at times $4*n$, $c_1 = 2$, $T_1 = 4$, $D_1 = 4$

τ_2 : periodic, available at times $4*n+1$, $c_2 = 1$, $T_1 = 4$, $D_2 = 1$

τ_1 has to start at $t = 0$

☞ deadline missed, but schedule is possible (start τ_2 first)

☞ scheduler is not optimal ☞ contradiction! q.e.d.



Scheduling without preemption

Preemption not allowed:  optimal schedules may leave processor idle to finish tasks with early deadlines arriving late.


 Knowledge about the future is needed for optimal scheduling algorithms

 No online algorithm can decide whether or not to keep idle.



EDF is optimal among all scheduling algorithms not keeping the processor idle at certain times.

If arrival times are known a priori, the scheduling problem becomes NP-hard in general. B&B typically used.

Summary

- Hard vs. soft deadlines
- Static vs. dynamic  TT-OS
- Schedulability

Classical scheduling

- Aperiodic tasks
 - No precedences
 - Simultaneous ( EDD)
& Asynchronous Arrival Times ( EDF, LL)
 - No preemption (brief)