Sharif University of Technology
Department of Computer Science and Engineering

Lec. 6:

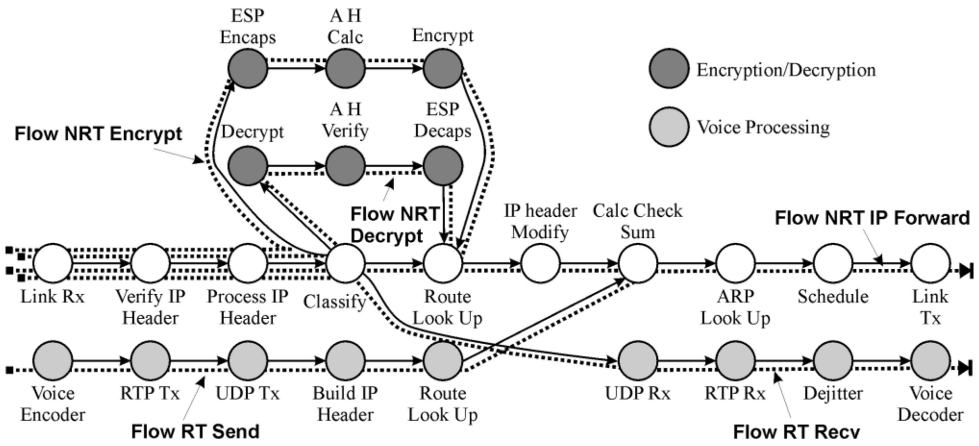
Mapping and Scheduling

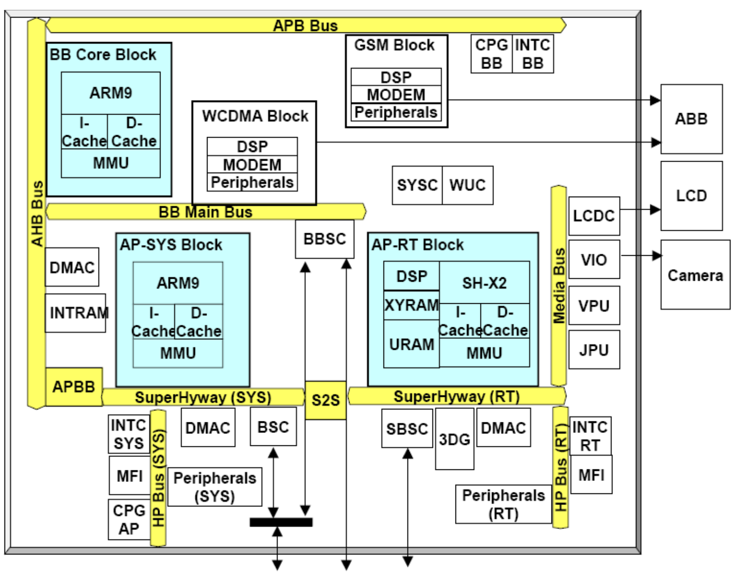
M. Ansari

Fall 2023

According to Peter Marwedel's Lectures

Mapping of Applications to Platforms





Distinction between mapping problems

	Embedded	PC-like
Architectures	Frequently heterogeneous very compact	Mostly homogeneous not compact (x86 etc)
x86 compatibility	Less relevant	Very relevant
Architecture fixed?	Sometimes not	Yes
Model of computation (MoCs)	C+multiple models (data flow, discrete events, ...)	Mostly von Neumann (C, C++, Java)
Optim. objectives	Multiple (energy, size, ...)	Average performance dominates
Real-time relevant	Yes, very!	Hardly
Applications	Several concurrent apps.	Mostly single application
Apps. known at design time	Most, if not all	Only some (e.g. WORD)

Problem Description

Given

- A set of applications
- Scenarios on how these applications will be used
- A set of candidate architectures comprising
  - (Possibly heterogeneous) processors
  - (Possibly heterogeneous) communication architectures
  - Possible scheduling policies

Tools urgently needed!

Find

- A mapping of applications to processors
- Appropriate scheduling techniques (if not fixed)
- A target architecture (if DSE is included)

Objectives

- Keeping deadlines and/or maximizing performance
- Minimizing cost, energy consumption

Related Work

- Mapping to ECUs in automotive design
- Scheduling theory:
  Provides insight for the mapping *task* $\rightarrow$ *start times*
- Hardware/software partitioning:
  Can be applied if it supports multiple processors
- High performance computing (HPC)
  Automatic parallelization, but only for
  - single applications,
  - fixed architectures,
  - no support for scheduling,
  - memory and communication model usually different
- High-level synthesis
  Provides useful terms like scheduling, allocation, assignment
- Optimization theory

Scope of mapping algorithms

Useful terms from hardware synthesis:

- **Resource Allocation**
  Decision concerning type and number of available resources
- **Resource Assignment**
  Mapping: Task → (Hardware) Resource
- **xx to yy binding:**
  Describes a mapping from behavioral to structural domain, e.g. task to processor binding, variable to memory binding
- **Scheduling**
  Mapping: Tasks → Task start times
  Sometimes, resource assignment is considered being included in scheduling.

Classes of mapping algorithms considered in this course

- **Classical scheduling algorithms**
  Mostly for independent tasks & ignoring communication, mostly for homogeneous multiprocessors
- **Dependent tasks as considered in architectural synthesis**
  Initially designed in different context, but applicable
- **Hardware/software partitioning**
  Dependent tasks, heterogeneous systems,
  focus on resource assignment
- **Design Space Exploration (DSE) using evolutionary algorithms;** Heterogeneous systems, incl. communication modeling

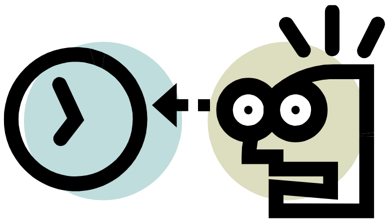
Real-time scheduling

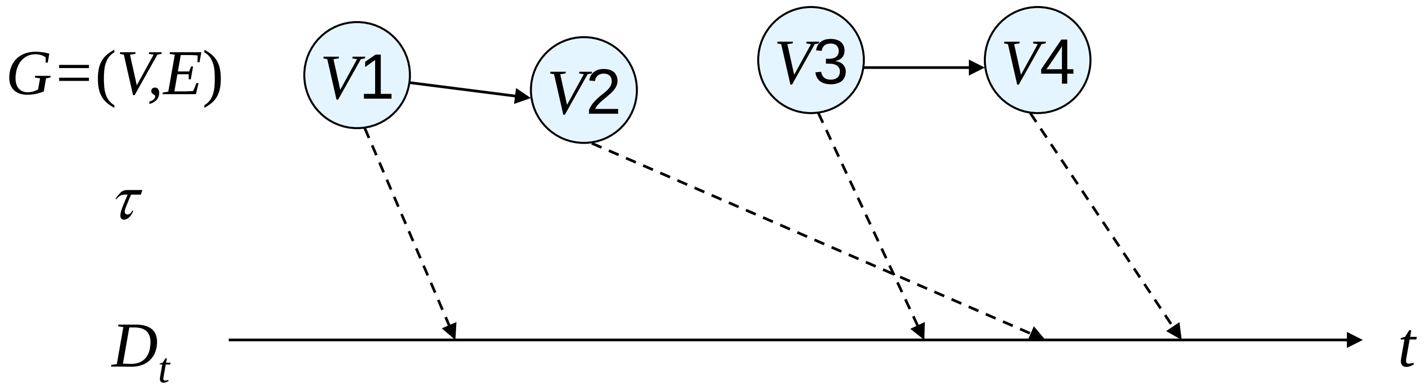
Assume that we are given a task graph $G=(V,E)$.

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

$$V \rightarrow D_t$$

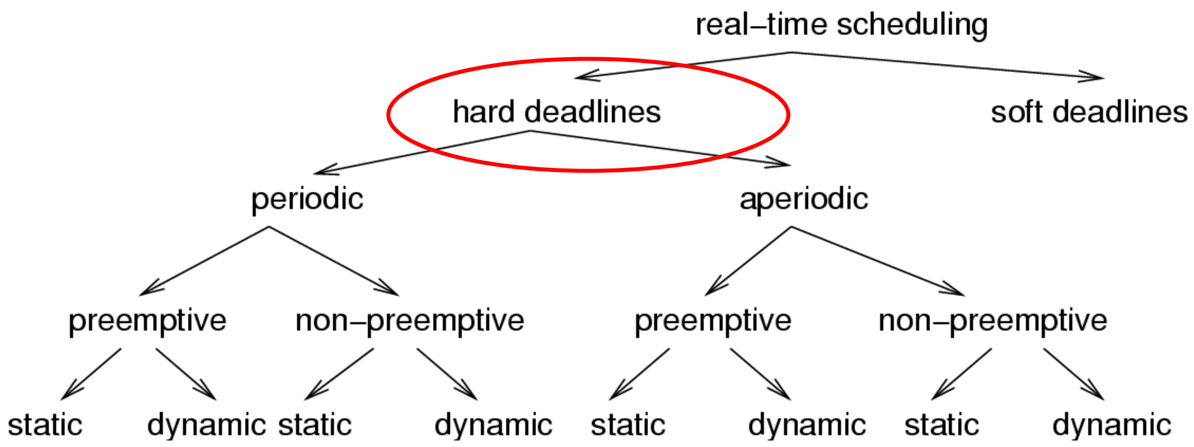
of a set of tasks V to start times from domain D_t.

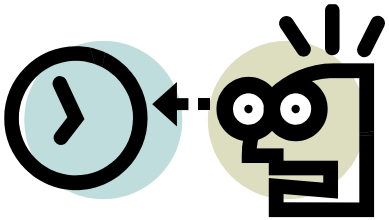




Typically, schedules have to respect a number of constraints, incl. resource constraints, dependency constraints, deadlines.
Scheduling = finding such a mapping.

Hard and soft deadlines



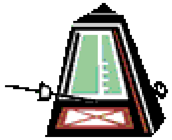


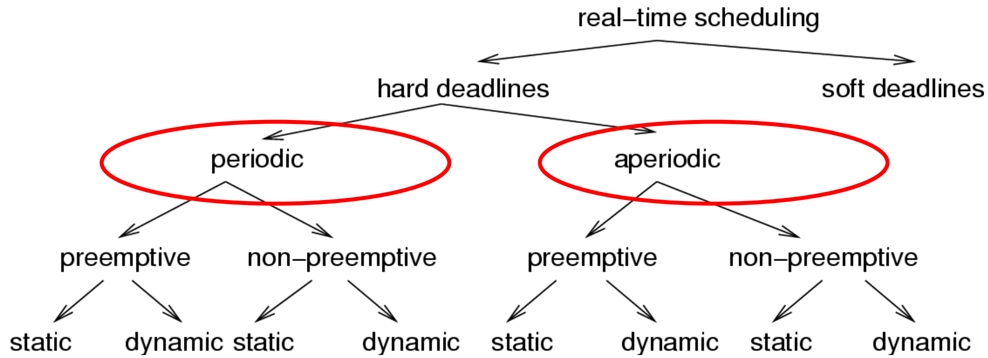
Def.: A time-constraint (deadline) is called **hard** if not meeting that constraint could result in a catastrophe [Kopetz, 1997].

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

We will focus on hard deadlines.

Periodic and aperiodic tasks



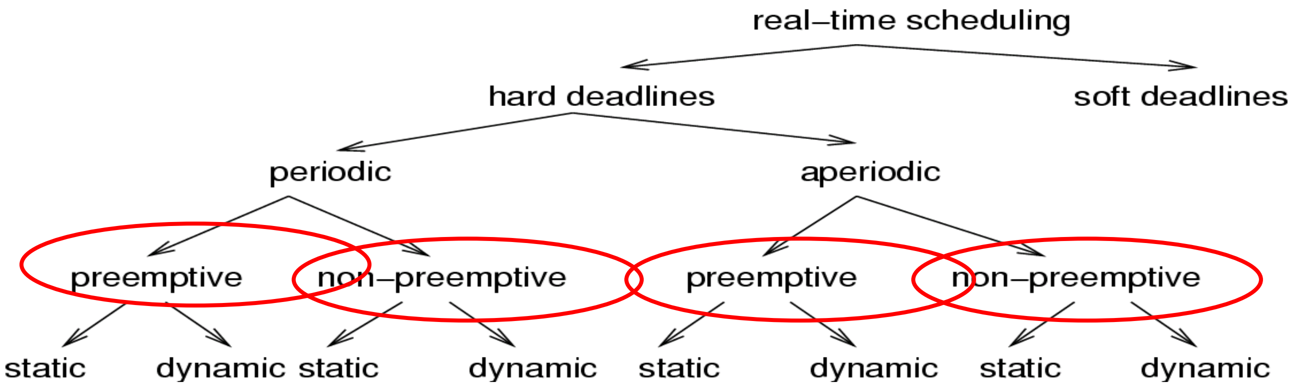


Def.: Tasks which must be executed once every *p* units of time are called **periodic** tasks. *p* is called their period. Each execution of a periodic task is called a **job**.

All other tasks are called **aperiodic**.

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.

Preemptive and non-preemptive scheduling



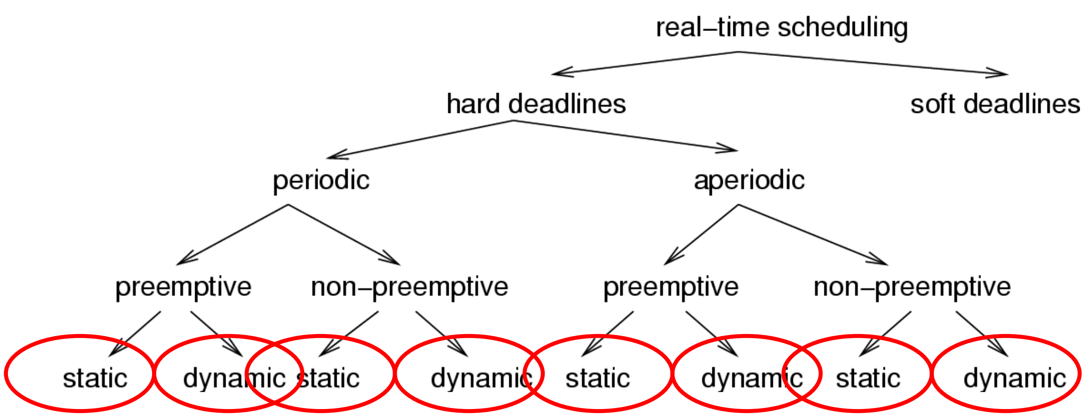
- **Non-preemptive schedulers:**

  Tasks 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

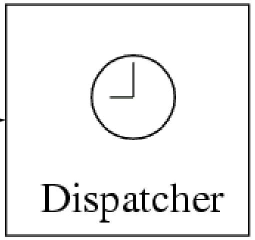
- **Dynamic/online scheduling:** Processor allocation decisions (scheduling) at run-time; based on the information about the tasks arrived so far.



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	start T1	12
17	send M5	
22	stop T1	
38	start T2	20
47	send M3	



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	Action	WCET
10	start T1	12
17	send M5	
22	stop T1	
38	start T2	20
47	send M3	

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

- **Single- 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.

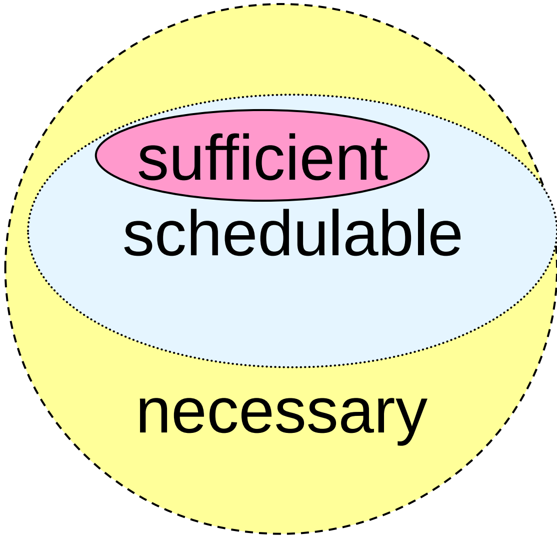
Schedulability

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. (Hopefully) small probability of not guaranteeing a schedule even though one exists.

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



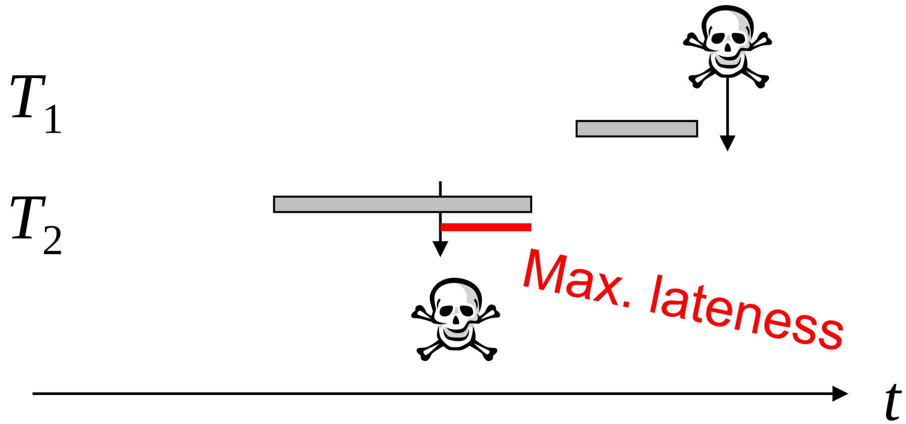
Cost functions

Cost function: Different algorithms aim at minimizing different functions.

Def.: Maximum lateness =

max$_{\textbf{all tasks}}$ (completion time – deadline)

Is <0 if all tasks complete before deadline.



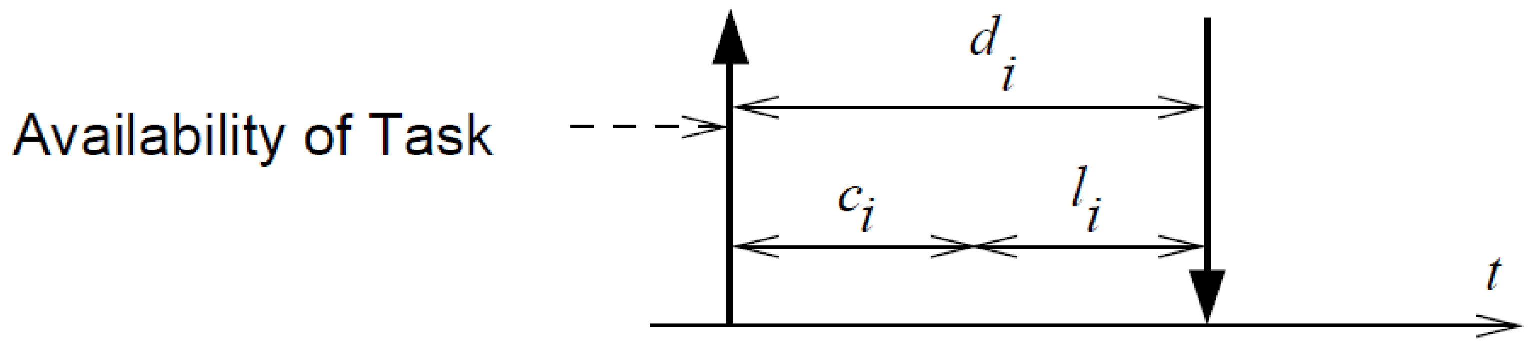
Classical scheduling algorithms for aperiodic systems

Graphics: © Alexandra Nolte, Gesine Marwedel, 2003

Aperiodic scheduling:
- Scheduling with no precedence constraints -

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

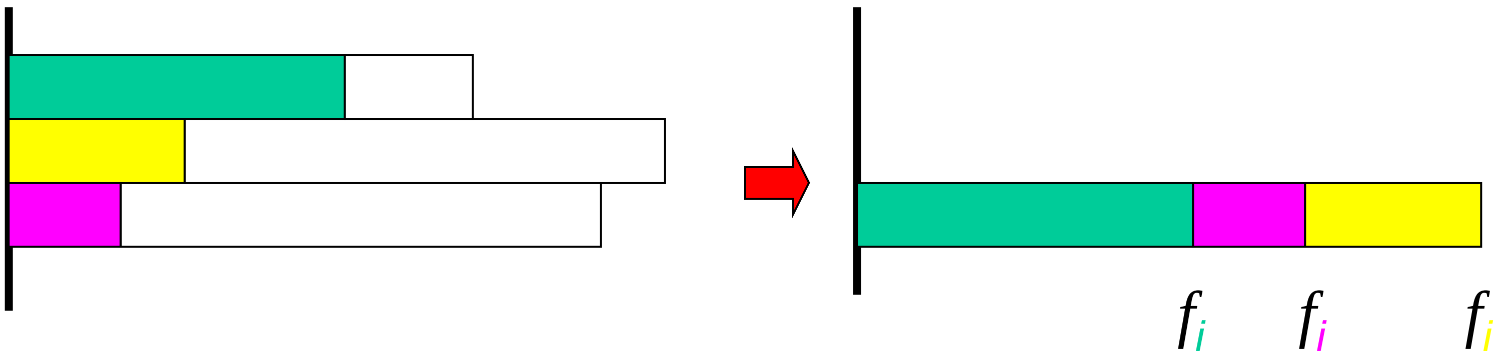
- c_i be the execution time of T_i ,
- d_i be the **deadline interval**, that is,
  the time between T_i becoming available
  and the time until which T_i has to finish execution.
- l_i be the **laxity** or **slack**, defined as $l_i = d_i - c_i$
- f_i be the finishing time.



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 $O(n \log(n))$.

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

Different arrival times: Preemption potentially reduces lateness.

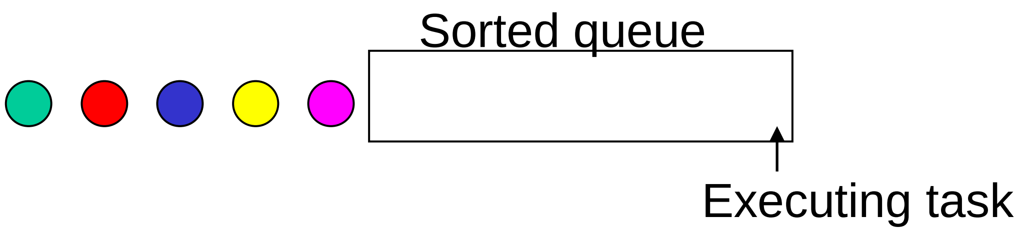
Theorem [Horn74]: Given a set of n independent tasks with arbitrary arrival times, any algorithm that at any instant executes the task with the earliest absolute deadline among all the ready tasks is optimal with respect to minimizing the maximum lateness.

Earliest Deadline First (EDF): - Algorithm -

Earliest deadline first (EDF) algorithm:

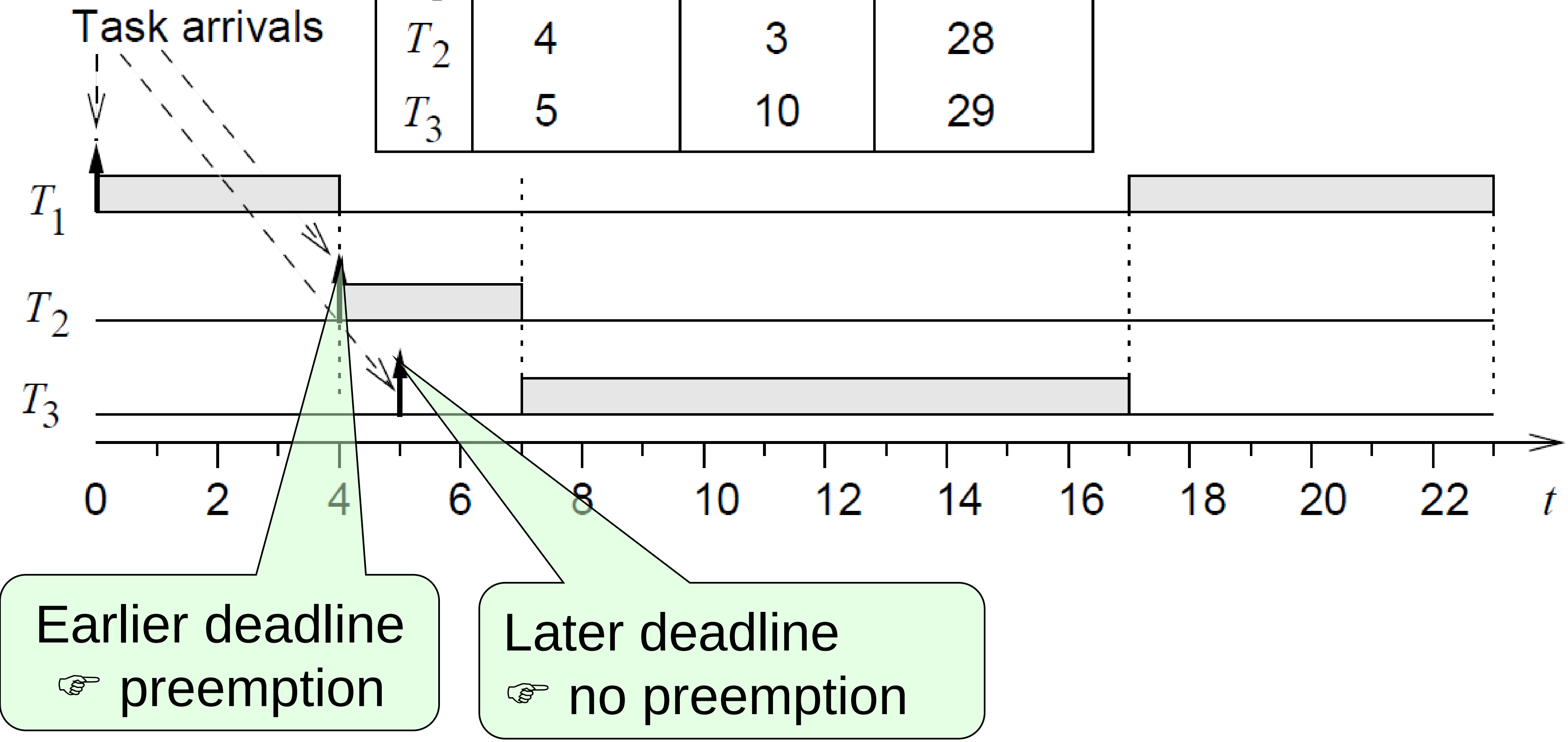
- Each time a new ready task arrives:
- It is inserted into a queue of ready tasks, sorted by their **absolute** deadlines. Task at head of queue is executed.
- If a newly arrived task is inserted at the head of the queue, the currently executing task is preempted.

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 -

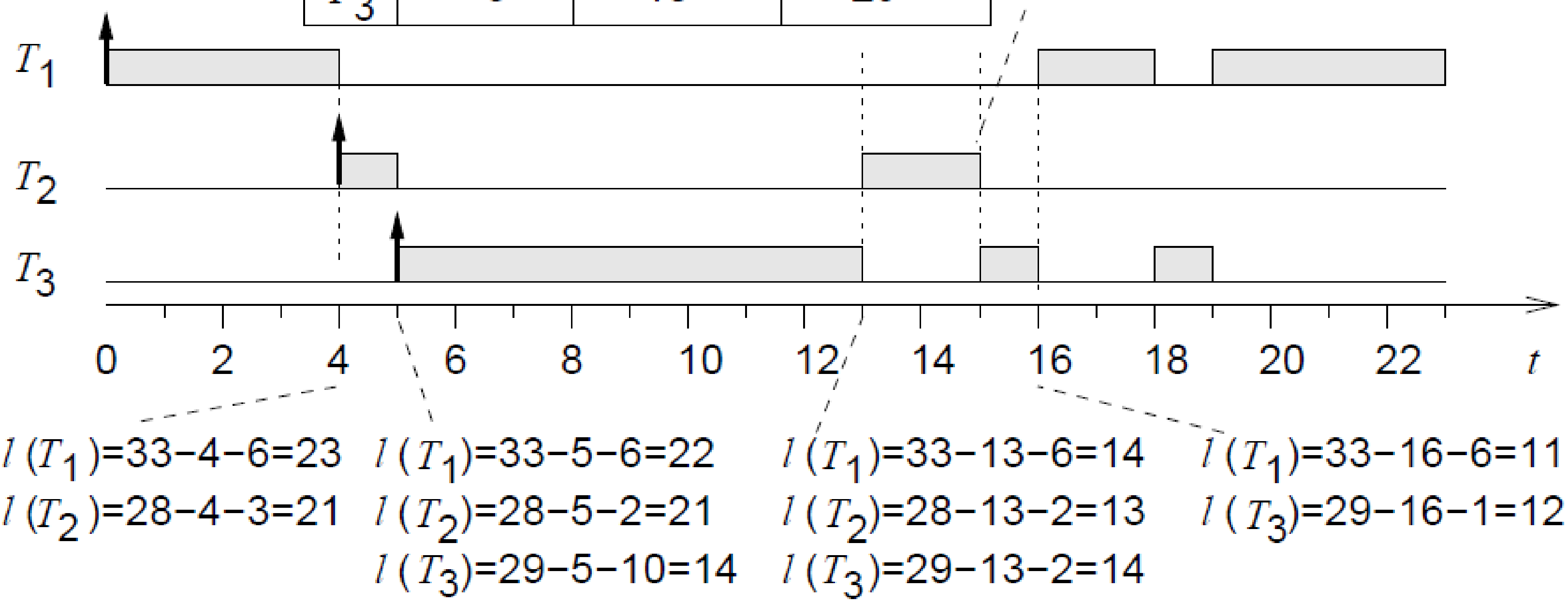
	arrival	duration	deadline
T_1	0	10	33
T_2	4	3	28
T_3	5	10	29



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

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

	arrival	duration	deadline
T_1	0	10	33
T_2	4	3	28
T_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.
- Dynamic priorities ☞ cannot be used with a fixed prio OS.
- LL scheduling requires the knowledge of the execution time.

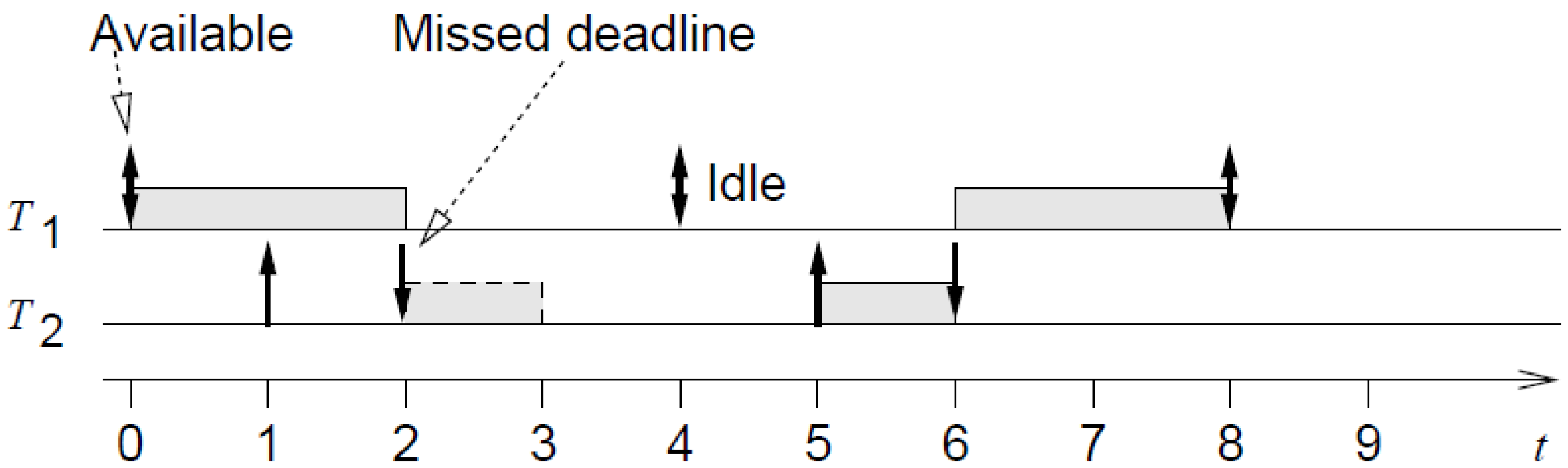
Scheduling without preemption (1)

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)

T_1: periodic, $c_1 = 2$, $p_1 = 4$, $d_1 = 4$
T_2: occasionally available at times $4*n+1$, $c_2= 1$, $d_2= 1$
T_1 has to start at t =0
☞ deadline missed, but schedule is possible (start T_2 first)
☞ scheduler is not optimal ☞ contradiction!



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. Branch & Bound typically used.

Summary

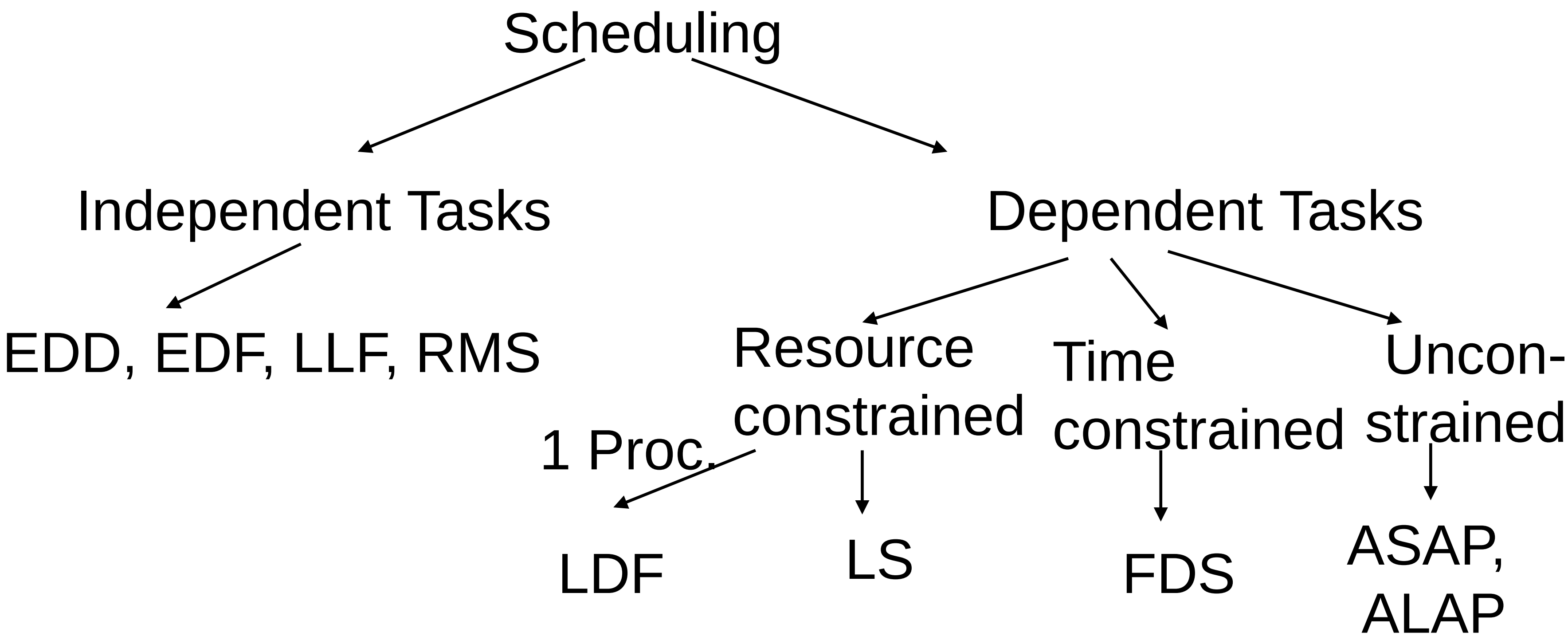
Definition mapping terms

- Resource allocation, assignment, binding, scheduling
- Hard vs. soft deadlines
- Static vs. dynamic ☞TT-OS
- Schedulability

Classical scheduling

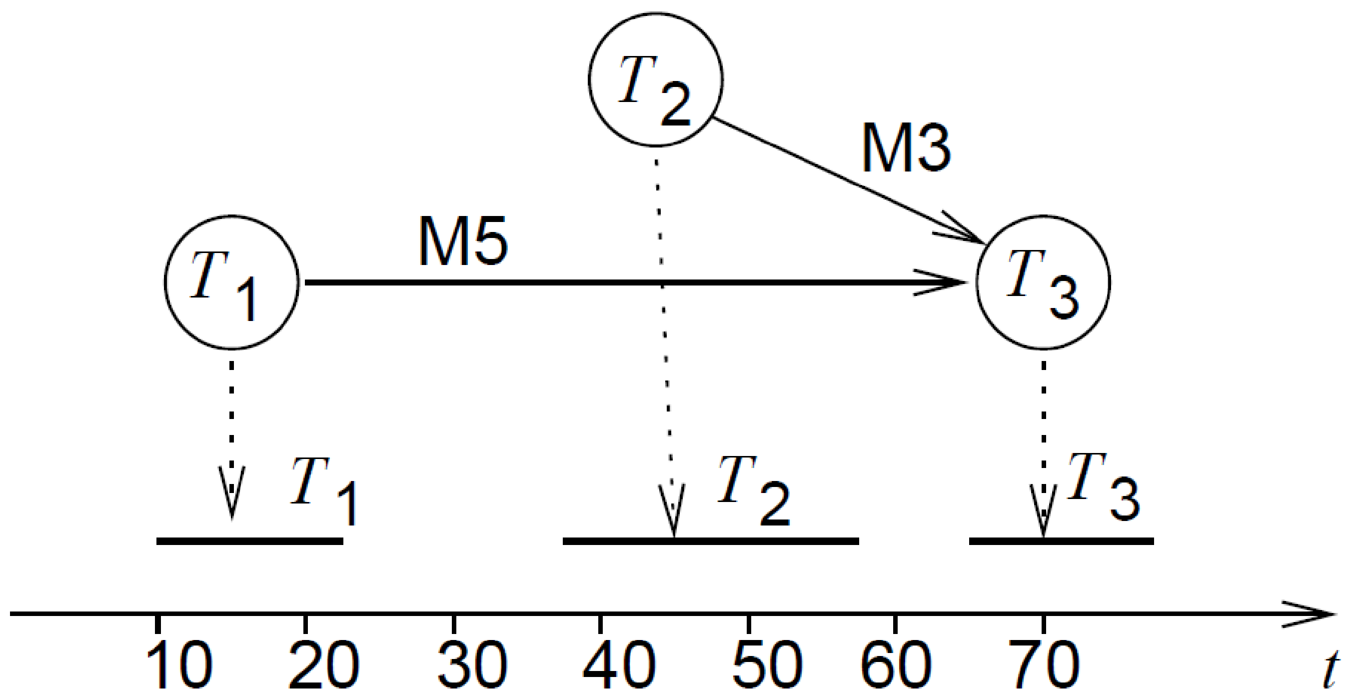
- Aperiodic tasks
  - No precedences
    - Simultaneous (☞EDD)
      & Asynchronous Arrival Times (☞EDF, LL)

Classification of Scheduling Problems



Scheduling with precedence constraints

Task graph and possible schedule:

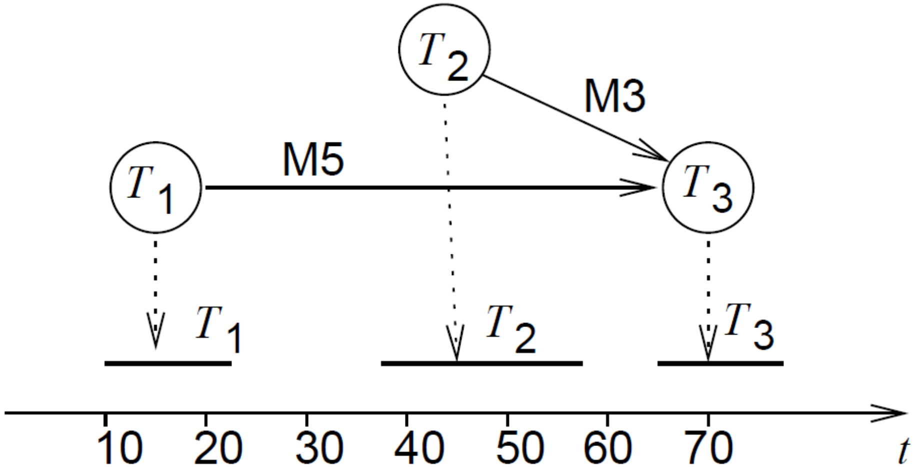


Simultaneous Arrival Times: The Latest Deadline First (LDF) Algorithm

LDF [Lawler, 1973]: reads the task graph and among the tasks with no successors inserts the one with the latest deadline into a queue. It then repeats this process, putting tasks whose successor have all been selected into the queue.

At run-time, the tasks are executed in the generated total order.

LDF is non-preemptive and is optimal for mono-processors.



If no local deadlines exist, LDF performs just a topological sort.

Asynchronous Arrival Times: Modified EDF Algorithm

This case can be handled with a modified EDF algorithm. The key idea is to transform the problem from a given set of dependent tasks into a set of independent tasks with different timing parameters [Chetto90].
This algorithm is optimal for mono-processor systems.

If preemption is not allowed, the heuristic algorithm developed by Stankovic and Ramamritham can be used.

Dependent tasks

The problem of deciding whether or not a schedule exists for a set of dependent tasks and a given deadline is NP-complete in general [Garey/Johnson].

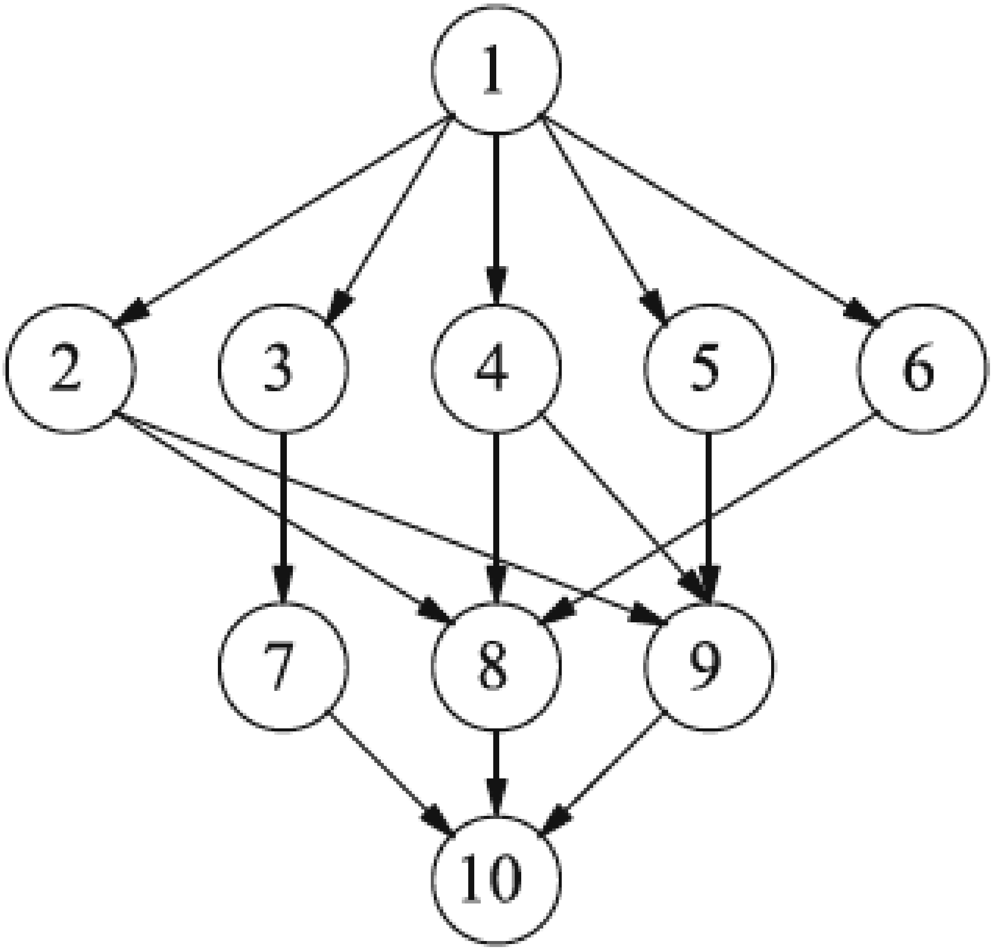
Strategies:

1. Add resources, so that scheduling becomes easier
2. Split problem into static and dynamic part so that only a minimum of decisions need to be taken at run-time.
3. Use scheduling algorithms from high-level synthesis

Classes of mapping algorithms considered in this course

- **Classical scheduling algorithms**
  Mostly for independent tasks & ignoring communication, mostly for mono- and homogeneous multiprocessors
- **Dependent tasks as considered in architectural synthesis**
  Initially designed in different context, but applicable
- **Hardware/software partitioning**
  Dependent tasks, heterogeneous systems, focus on resource assignment
- **Design space exploration using genetic algorithms**
  Heterogeneous systems, incl. communication modeling

Task graph



Task	C_i
1	9
2	13
3	11
4	8
5	10
6	9
7	7
8	5
9	12
10	7

As soon as possible (ASAP) scheduling

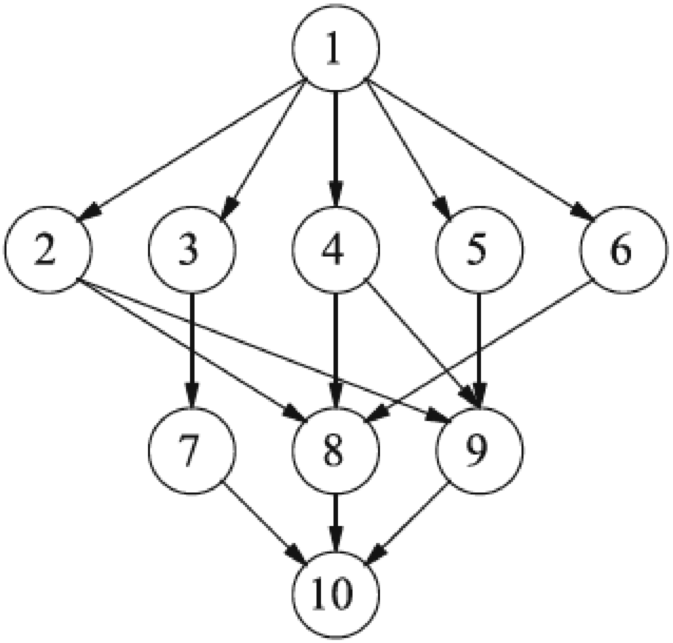
ASAP: All tasks are scheduled as early as possible

Loop over (integer) time steps:

- Compute the set of unscheduled tasks for which all predecessors have finished their computation
- Schedule these tasks to start at the current time step.

As soon as possible (ASAP) scheduling: Example (1)

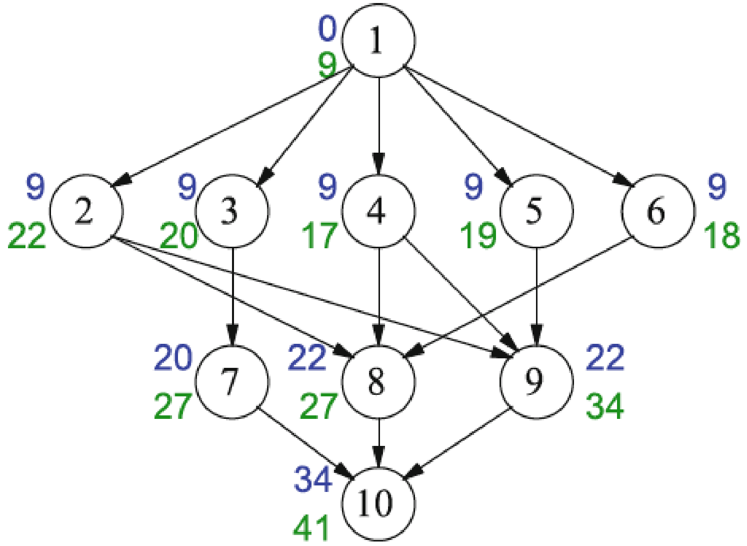
```
for (t=0; there are unscheduled tasks; t++)   {
   τ'={all tasks for which all predecessors finished};
   set start time of all tasks in τ' to t;
}
```

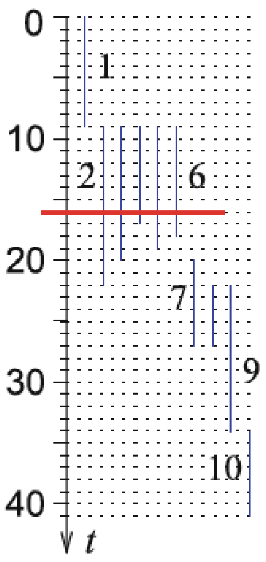


Task	C_i
1	9
2	13
3	11
4	8
5	10
6	9
7	7
8	5
9	12
10	7

As soon as possible (ASAP) scheduling: Example (2)

```
for (t=0; there are unscheduled tasks; t++)   {
   τ'={all tasks for which all predecessors finished};
   set start time of all tasks in τ' to t;
}
```





As-late-as-possible (ALAP) scheduling

ALAP: All tasks are scheduled as late as possible

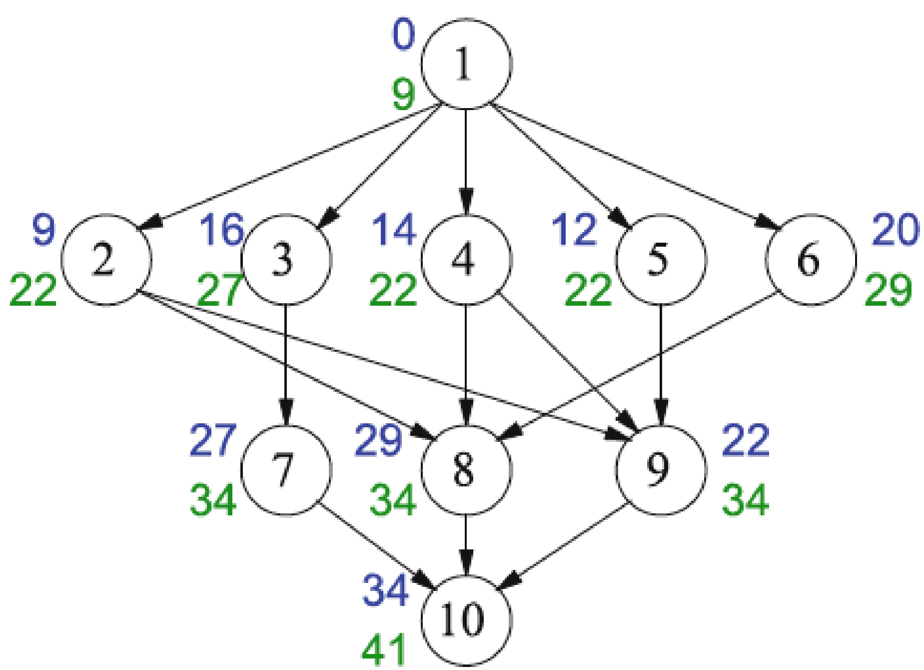
Start at last time step*:

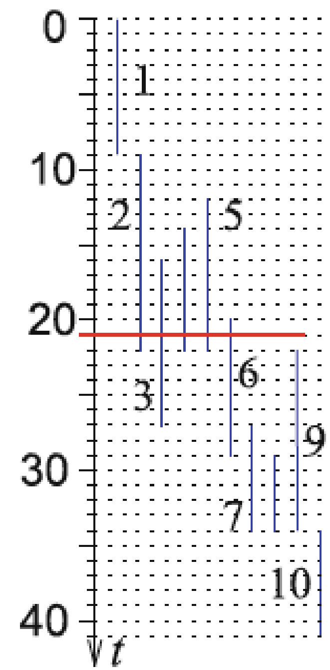
Schedule tasks with no successors and tasks for which all successors have already been scheduled.

* Generate a list, starting at its end

As-late-as-possible (ALAP) scheduling: Example

```
for (t=0; there are unscheduled tasks; t--) {
    τ'={all tasks on which no unscheduled task depends};
    set start time of all tasks in τ' to (t - their execution time);
}
Shift all times such that the first tasks start at t=0.
```





(Resource constrained) List Scheduling

List scheduling: extension of ALAP/ASAP method

Preparation:

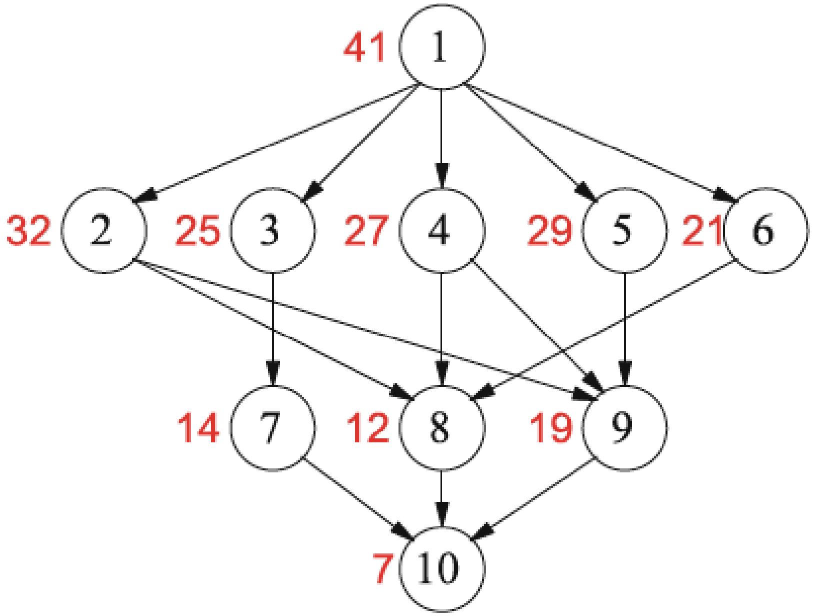
- Topological sort of task graph $G=(V,E)$
- Computation of priority of each task:

  Possible priorities u:

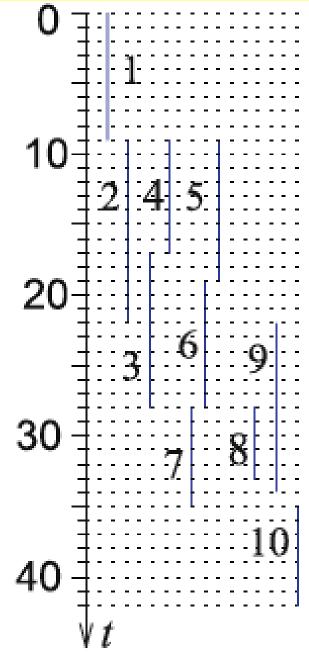
  - Number of successors
  - **Longest path**
  - **Mobility** = τ (ALAP schedule)- τ (ASAP schedule)

List Scheduling with Longest Path: Example

```
for (t=0; there are unscheduled tasks; t++)          /* loop over times */
  for (l ∈ L) {                                       /* loop over resource types */
    τ*_{t,l} = set of tasks of type l still executing at time t;
    τ**_{t,l} = set of tasks of type l ready to start execution at time t;
    Compute set τ'_t ⊆ τ**_{t,l} of maximum priority such that
    |τ'_t| + |τ*_{t,l}| ≤ B_l .
    Set start times of all τ_i ∈ τ'_t to t: s_i = t;
  }
```

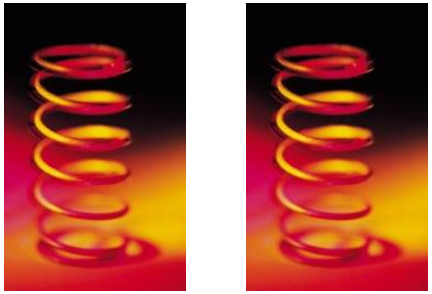


We have just three processor



(Time constrained) Force-directed scheduling

- Goal: balanced utilization of resources
- Based on spring model;
- Originally proposed for high-level synthesis



Pierre G. Paulin, J.P. Knight, Force-directed scheduling in automatic data path synthesis, *Design Automation Conference* (DAC), 1987, S. 195-202

Evaluation of HLS-Scheduling

- Focus on considering dependencies
- Mostly heuristics, few proofs on optimality
- Not using global knowledge about periods etc.
- Considering discrete time intervals
- Variable execution time available only as an extension
- Includes modeling of heterogeneous systems

Overview

Scheduling of tasks with real-time constraints:
Table with some known algorithms

	Equal arrival times; non-preemptive	Arbitrary arrival times; preemptive
Independent tasks	EDD (Jackson)	EDF (Horn)
Dependent tasks	LDF (Lawler), ASAP, ALAP, LS, FDS	EDF* (Chetto)

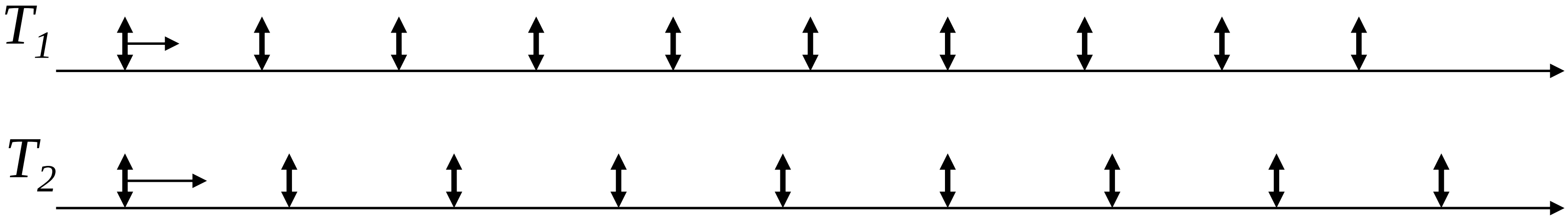
Conclusion

- HLS-based scheduling
  - ASAP
  - ALAP
  - *List scheduling* (LS)
  - *Force-directed scheduling* (FDS)
- Evaluation

Classes of mapping algorithms considered in this course

- **Classical scheduling algorithms**
  Mostly for independent tasks & ignoring communication, mostly for mono- and homogeneous multiprocessors
- **Dependent tasks as considered in architectural synthesis**
  Initially designed in different context, but applicable
- **Hardware/software partitioning**
  Dependent tasks, heterogeneous systems, focus on resource assignment
- **Design space exploration using evolutionary algorithms;** Heterogeneous systems, incl. communication modeling

Periodic scheduling



Each execution instance of a task is called a **job**.

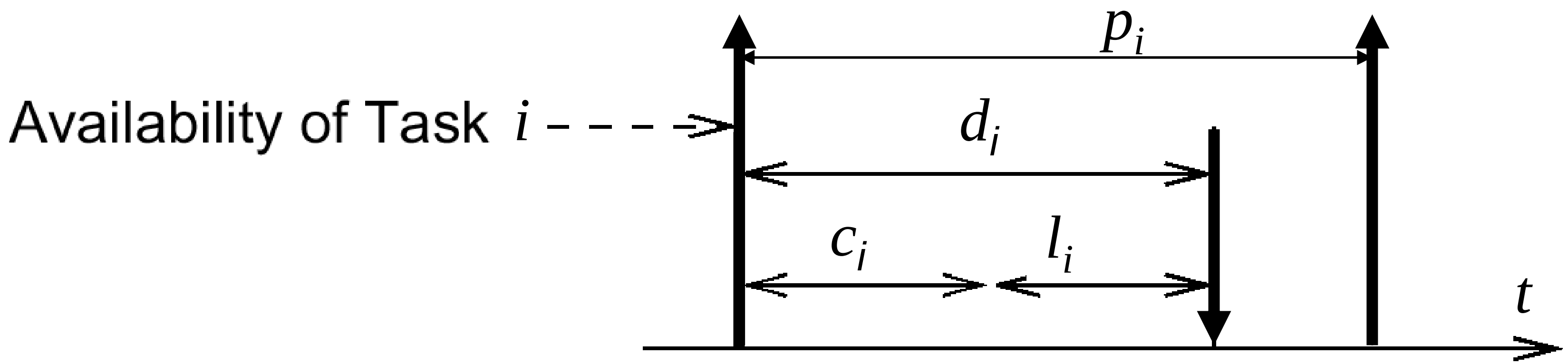
Notion of optimality for aperiodic scheduling does not make sense for periodic scheduling.

For periodic scheduling, the best that we can do is to design an algorithm which will always find a schedule if one exists.
☞ A scheduler is defined to be **optimal** iff it will find a schedule if one exists.

Periodic scheduling: Scheduling with no precedence constraints

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

- p_i be the period of task T_i,
- c_i be the execution time of T_i ,
- d_i be the **deadline interval**, that is,
  the time between T_i becoming available
  and the time until which T_i has to finish execution.
- l_i be the **laxity** or **slack**, defined as $l_i = d_i - c_i$
- f_i be the finishing time.



Average utilization: important characterization of scheduling problems

Average utilization: $$\mu = \sum_{i=1}^{n} \frac{c_i}{p_i}$$

Necessary condition for schedulability (with m=number of processors): $$\mu \leq m$$

Independent tasks: Rate monotonic (RM) scheduling

Most well-known technique for scheduling independent periodic tasks [Liu, 1973].

Assumptions:

- All tasks that have hard deadlines are periodic.
- All tasks are independent.
- $d_i = p_i$, for all tasks.
- c_i is constant and is known for all tasks.
- The time required for context switching is negligible.
- For a single processor and for n tasks, the following equation holds for the average utilization μ:

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

Rate monotonic (RM) scheduling
- The policy -

RM policy: **The priority of a task is a monotonically decreasing function of its period.**

At any time, a highest priority task among all those that are ready for execution is allocated.

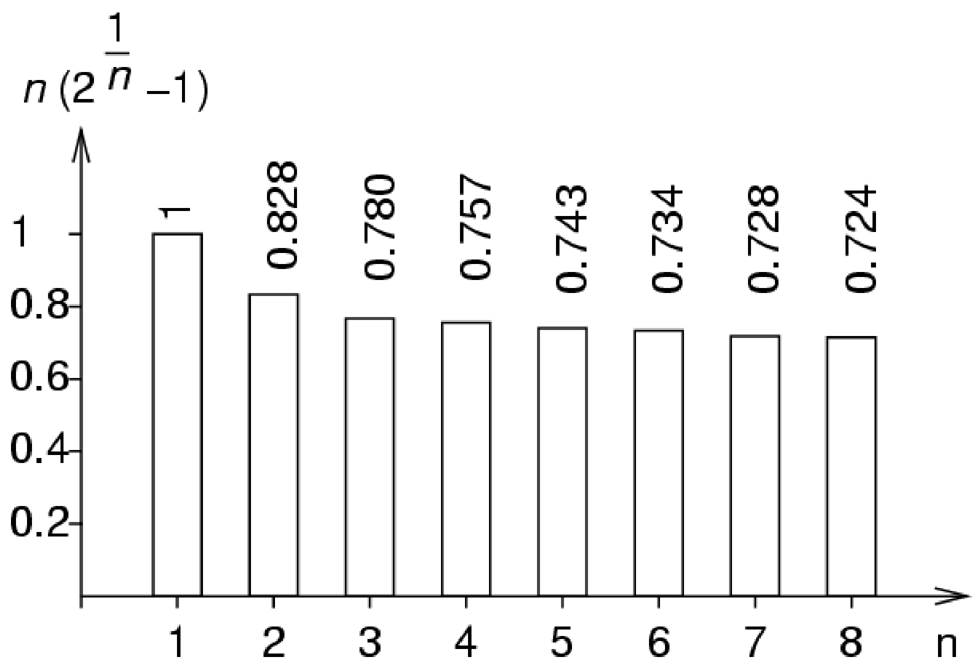
Theorem: If all RM assumptions are met, schedulability is guaranteed.

Maximum utilization for guaranteed schedulability

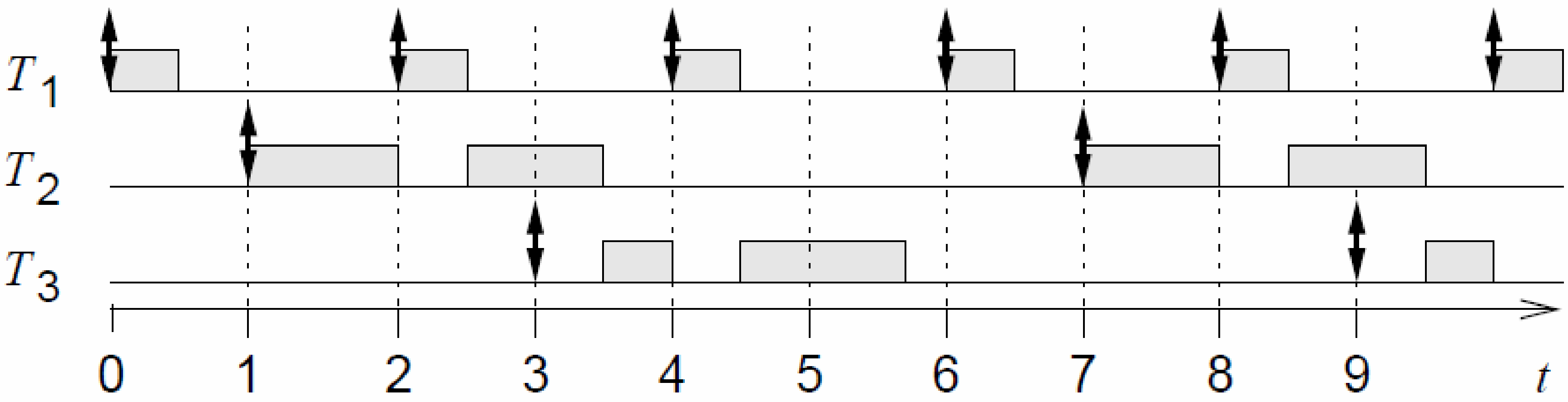
Maximum utilization as a function of the number of tasks:

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

$$\lim_{n \to \infty}(n(2^{1/n} - 1) = \ln(2)$$



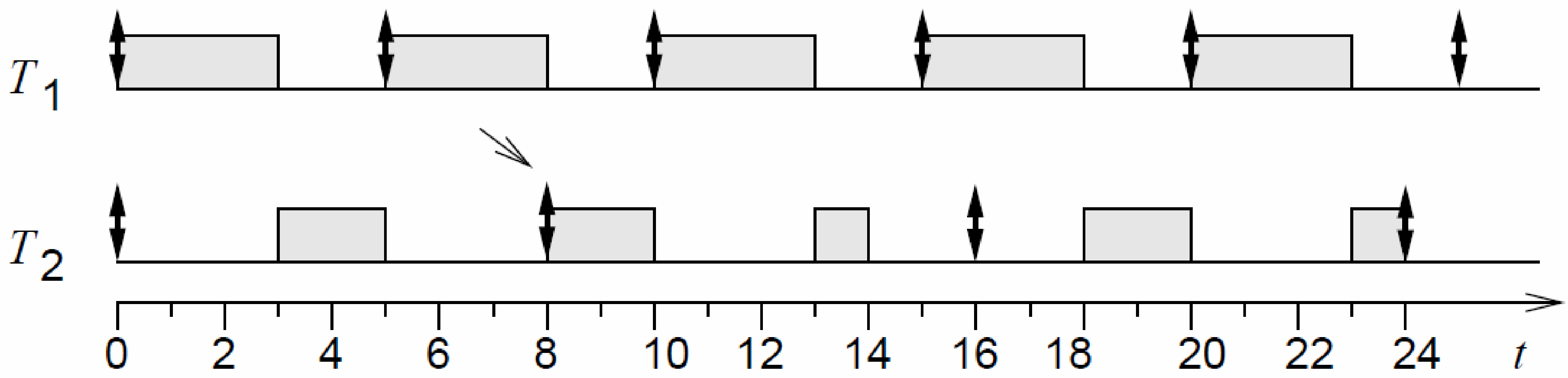
Example of RM-generated schedule



T_1 preempts T_2 and T_3.
T_2 and T_3 do not preempt each other.

Failing RMS

Task 1: period 5, execution time 3
Task 2: period 8, execution time 3
μ=3/5+3/8=24/40+15/40=39/40 ≈ 0.975
$2(2^{1/2}-1) \approx 0.828$



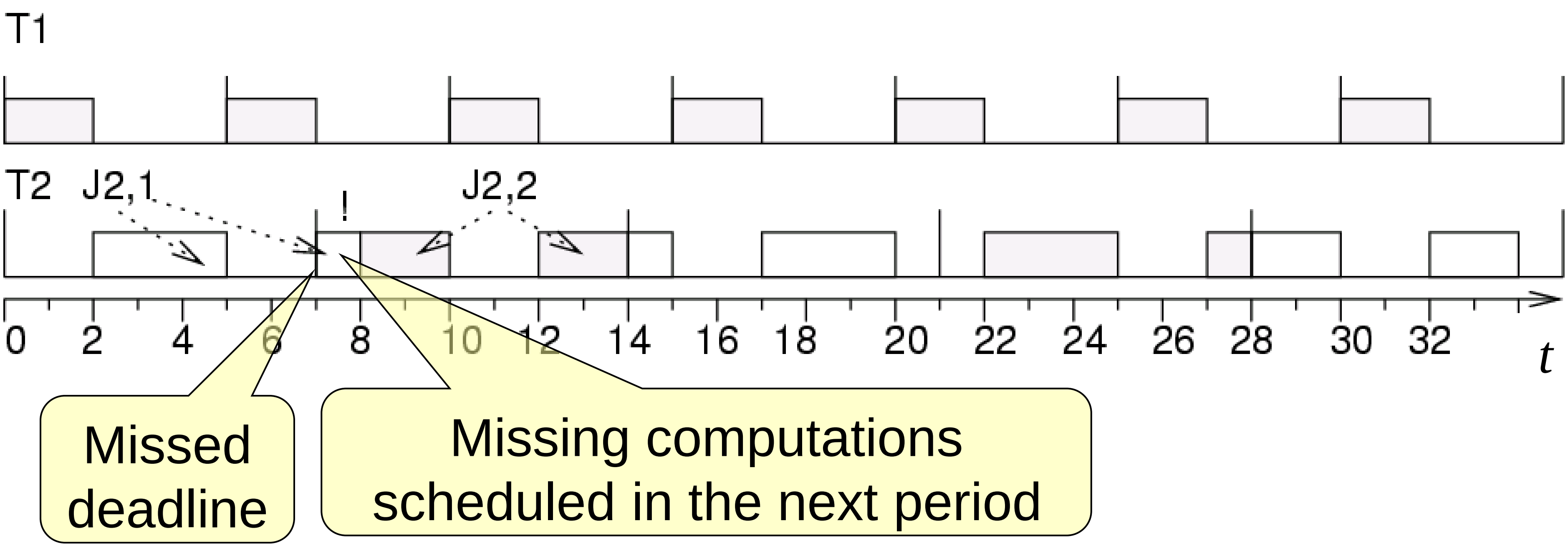
Case of failing RM scheduling

Task 1: period 5, execution time 2

Task 2: period 7, execution time 4

$\mu = 2/5 + 4/7 = 34/35 \approx 0.97$

$2(2^{1/2} - 1) \approx 0.828$



Properties of RM scheduling

- RM scheduling is based on **static** priorities. This allows RM scheduling to be used in an OS with static priorities, such as Windows NT.
- No idle capacity is needed if $\forall i$: $p_{i+1}=F\, p_i$:
  i.e. if the **period of each task is a multiple of the period of the next higher priority task**, schedulability is then also guaranteed if $\mu \leq 1$.
- A huge number of variations of RM scheduling exists.
- In the context of RM scheduling, many formal proofs exist.

EDF

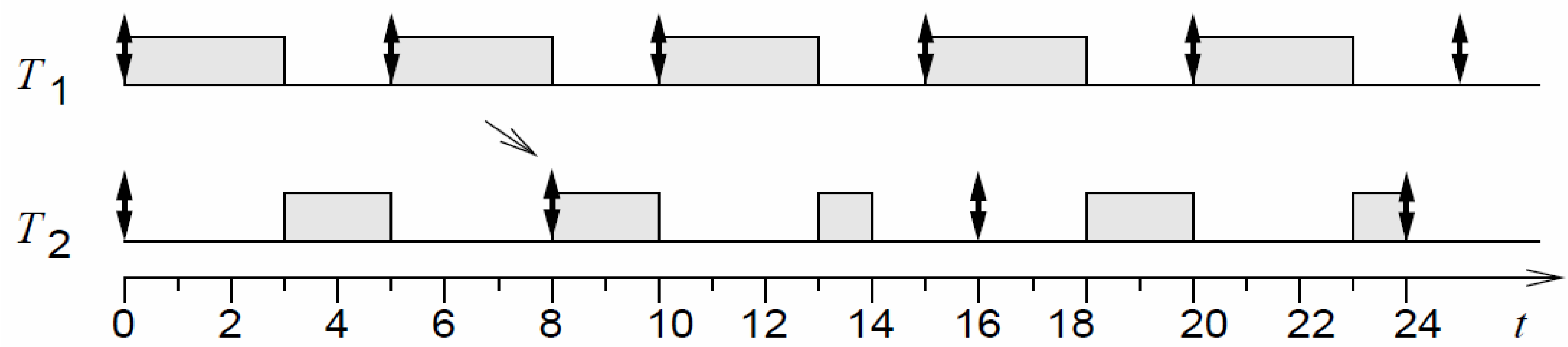
EDF can also be applied to periodic scheduling.

EDF optimal for every **hyper-period**
(= least common multiple of all periods)

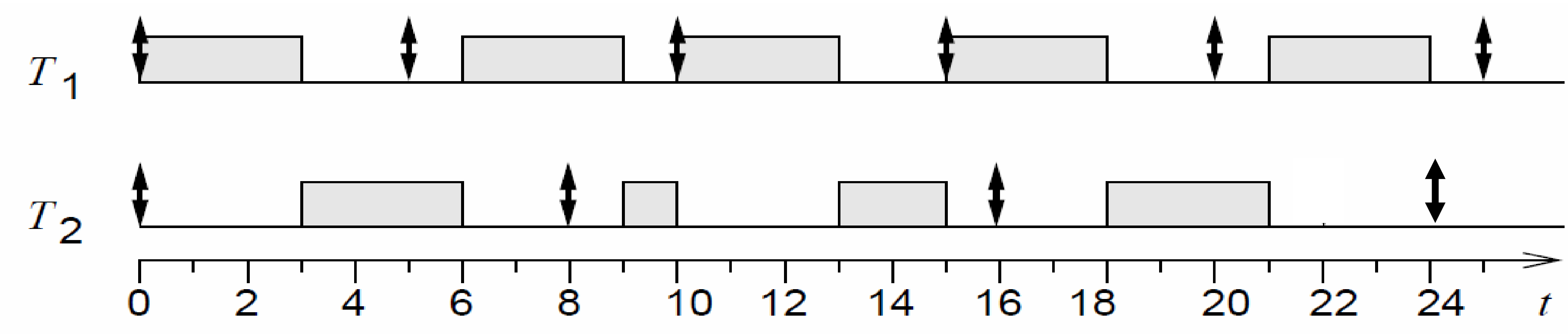
- ☞ Optimal for periodic scheduling
- ☞ EDF must be able to schedule the example in which RMS failed.

Comparison EDF/RMS

RMS:



EDF:



T_2 not preempted, due to its earlier deadline.

EDF: Properties

EDF requires dynamic priorities

☞ EDF cannot be used with an operating system just providing static priorities.

However, a recent paper (by Margull and Slomka) at DATE 2008 demonstrates how an OS with static priorities can be extended with a plug-in providing EDF scheduling
(key idea: delay tasks becoming ready if they shouldn't be executed under EDF scheduling.

Comparison RMS/EDF

	RMS	EDF
Priorities	Static	Dynamic
Works with OS with fixed priorities	**Yes**	**No***
Uses full computational power of processor	No, just up till $\mu=n(2^{1/n}-1)$	Yes
Possible to exploit full computational power of processor without provisioning for slack	**No**	**Yes**

* Unless the plug-in by Slomka et al. is added.

Sporadic tasks

If sporadic tasks were connected to interrupts, the execution time of other tasks would become very unpredictable.

☞ Introduction of a sporadic task server,
periodically checking for ready sporadic tasks;

☞ Sporadic tasks are essentially turned into periodic tasks.

Dependent tasks

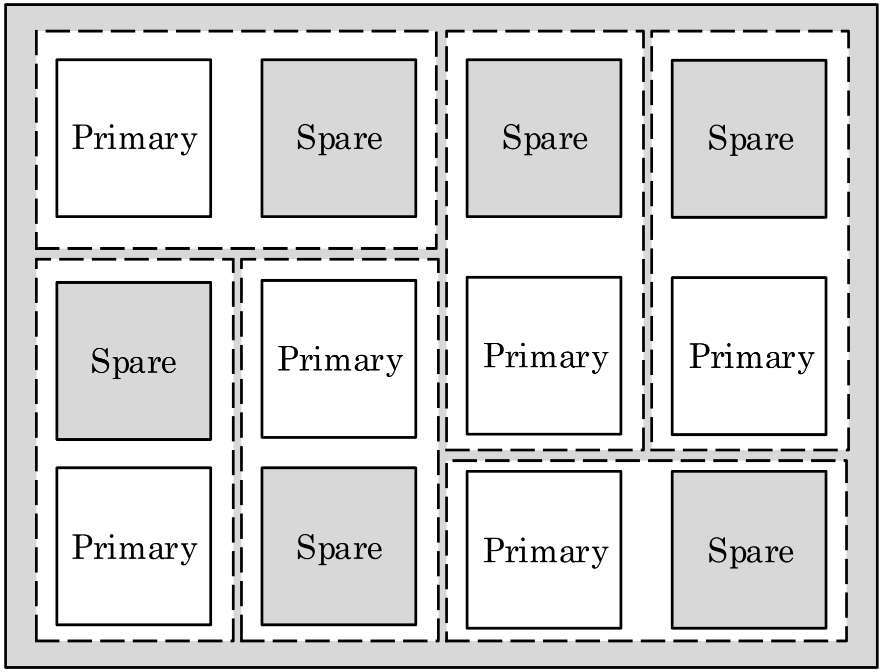
The problem of deciding whether or not a schedule exists for a set of dependent tasks and a given deadline is NP-complete in general [Garey/Johnson].

Strategies:

1. Add resources, so that scheduling becomes easier
2. Split problem into static and dynamic part so that only a minimum of decisions need to be taken at run-time.
3. Use scheduling algorithms from high-level synthesis

Power-Aware Scheduling in FT Embedded Systems

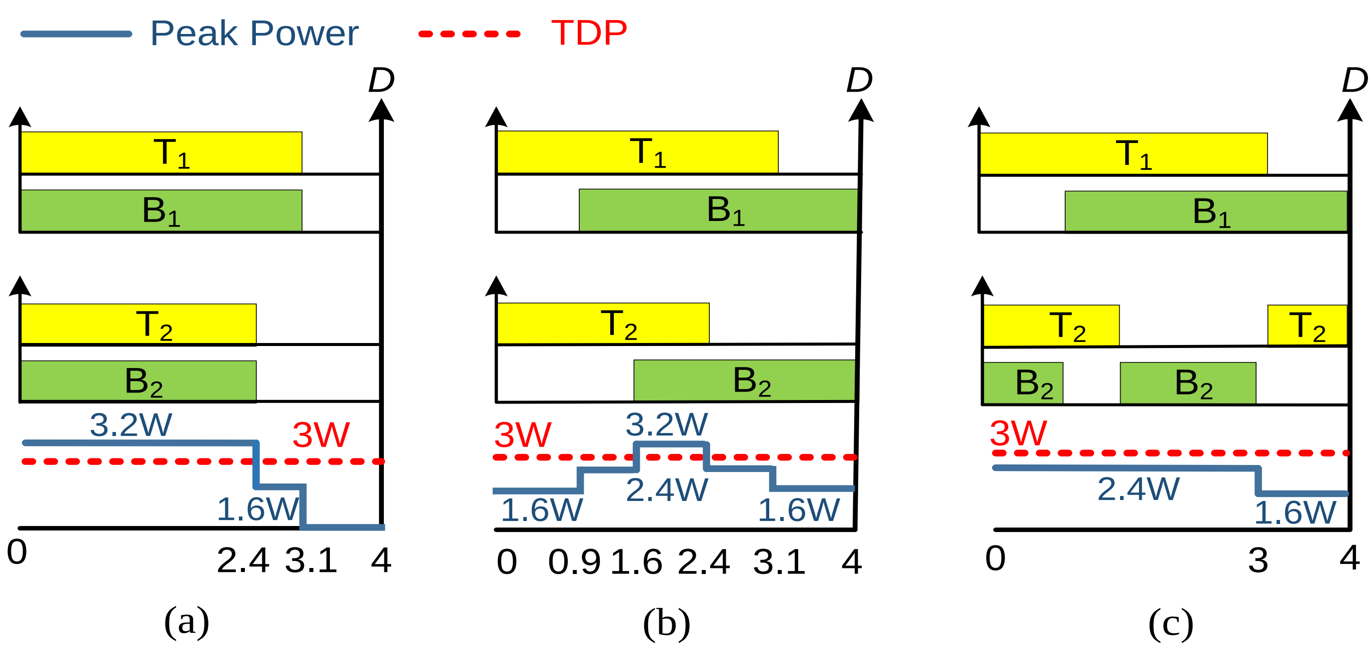
- Standby-sparing technique



Example of a multicore chip with 12 cores

Motivational Example

- T_1: QSORT and T_2: TIFF from Mibench benchmark suite



Motivational analysis of peak power problem in the primary-backup technique.

Power-Aware Scheduling (2)

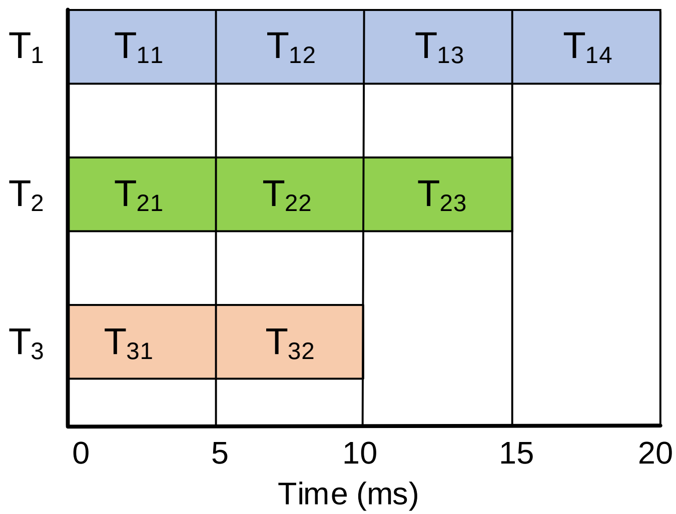
- Three main operations:
  - ***Power Tracing***
  - ***Task Partitioning***
  - ***Scheduling policies***

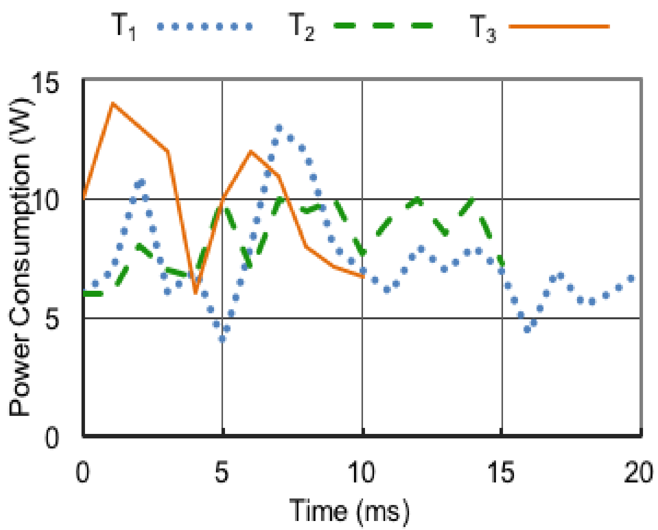
- There are three key challenges:
  - ***Thermal Design Power constraint***
  - ***Timing constraints***
  - ***Reliability***

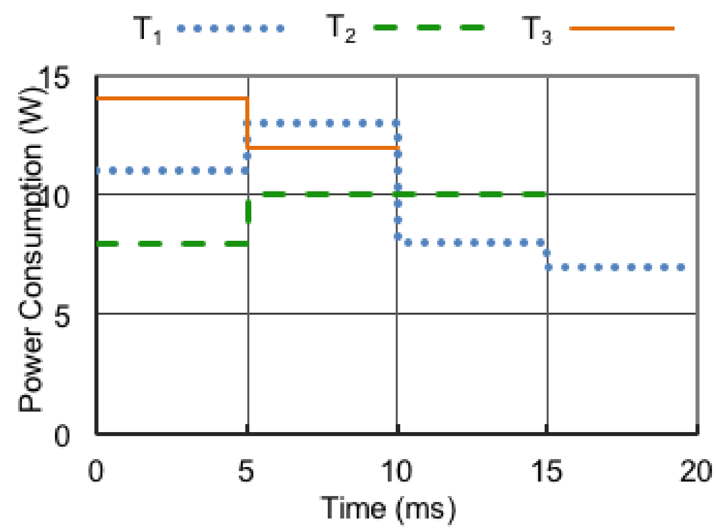
Illustrate Example

- Consider three tasks T_1, T_2 and T_3 with the deadline d_i and worst-case execution time WC_i:
  - $WC_1=20$, $WC_2=15$, $WC_3=10$, $d_1=d_2=d_3=60$
  - $PP_{max1}=13$, $PP_{max2}=10$, $PP_{max3}=14$
  - Partitioning Slot (*PS*) = 5
  - A chip with two cores {C1, C2}
  - TDP value = 20 W

Dividing the tasks into the parts

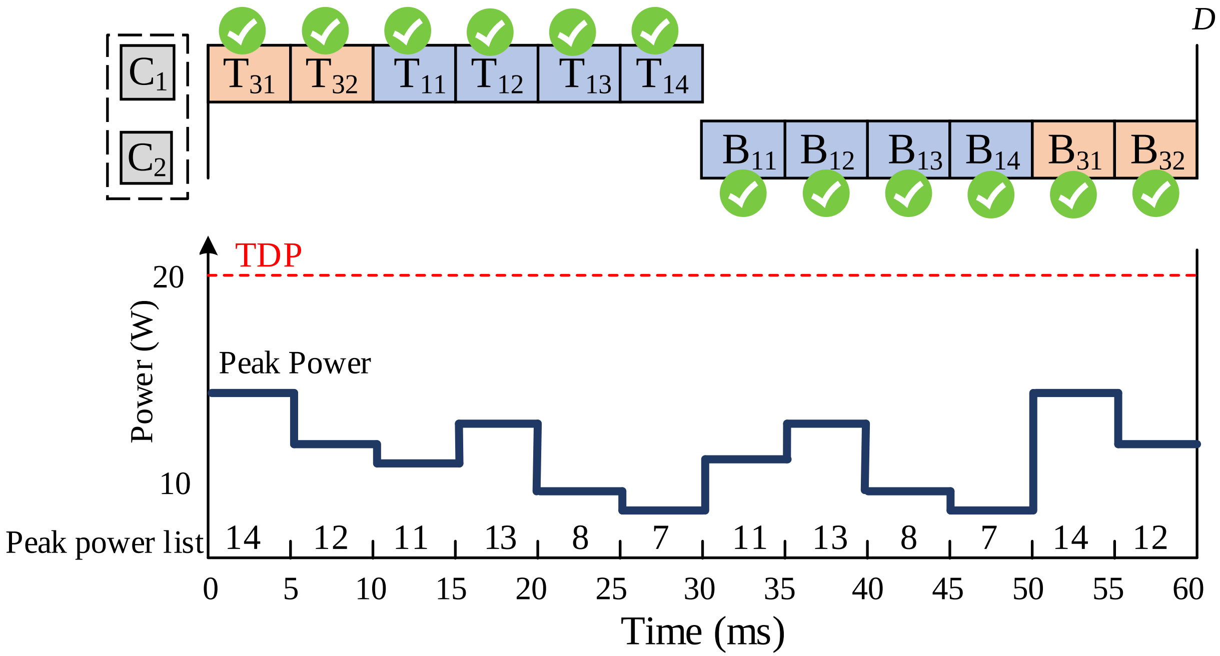






Offline Phase

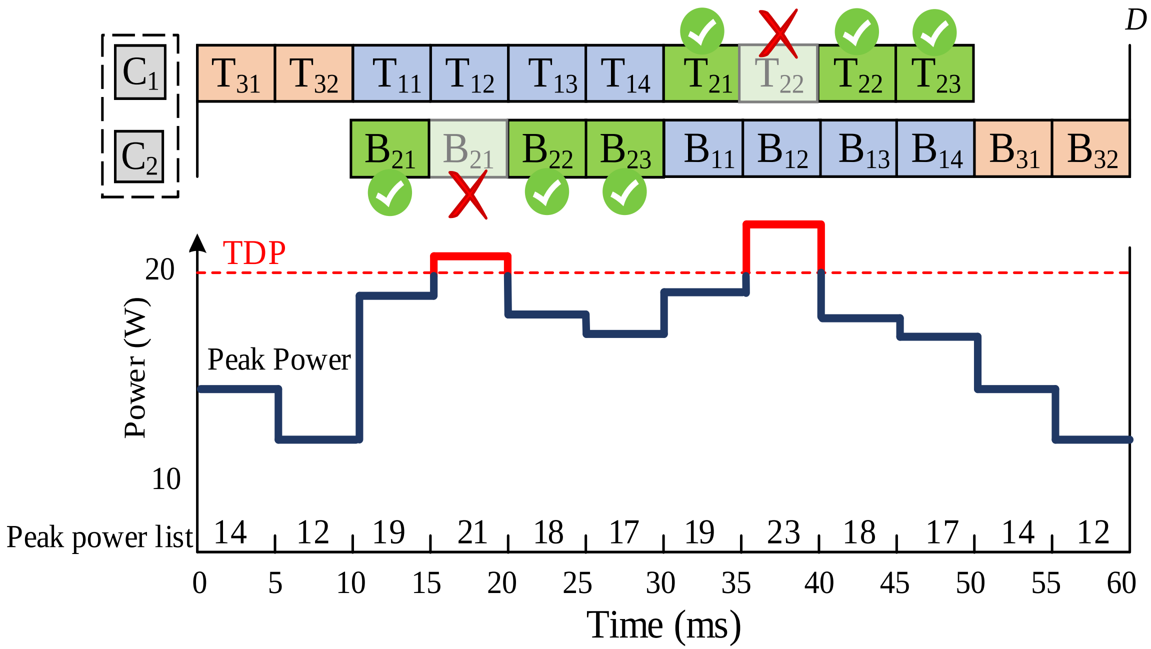
- Proposal: The RAPPM scheduling algorithm
  - The Maximum-Peak-Power-First policy on primary cores
  - The Maximum-Peak-Power-Last policy on spare cores



The RAPPM scheduling on a core pair

Offline Phase (Cont'd)

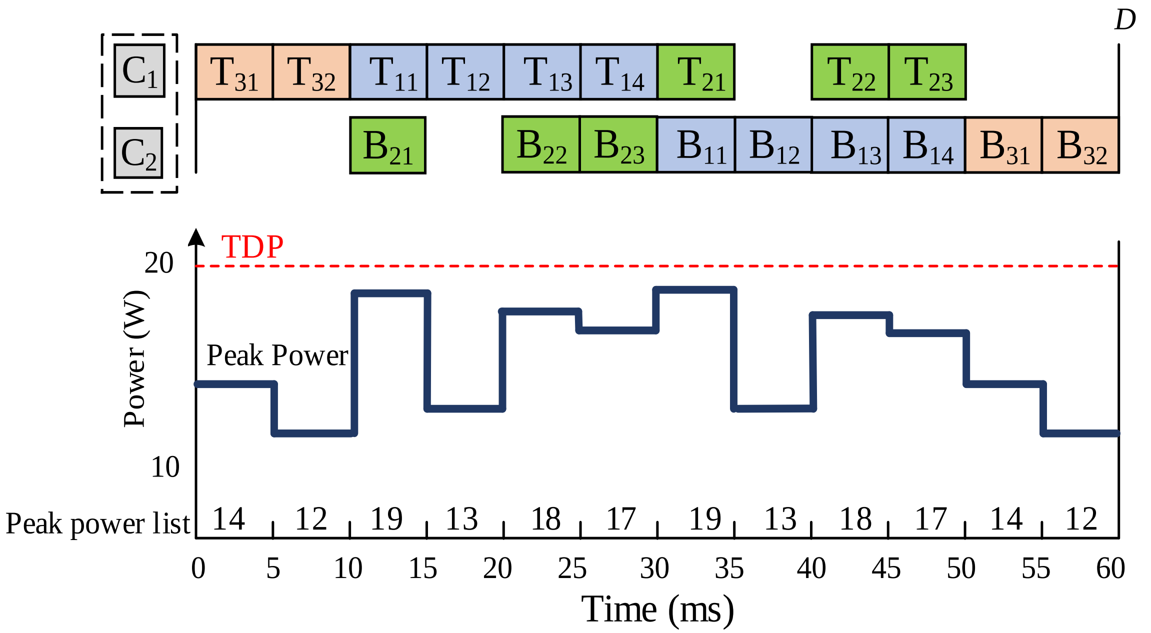
- Proposal: The RAPPM scheduling algorithm
  - The Maximum-Peak-Power-First policy on primary cores
  - The Maximum-Peak-Power-Last policy on spare cores



The RAPPM scheduling on a core pair

Offline Phase (Cont'd)

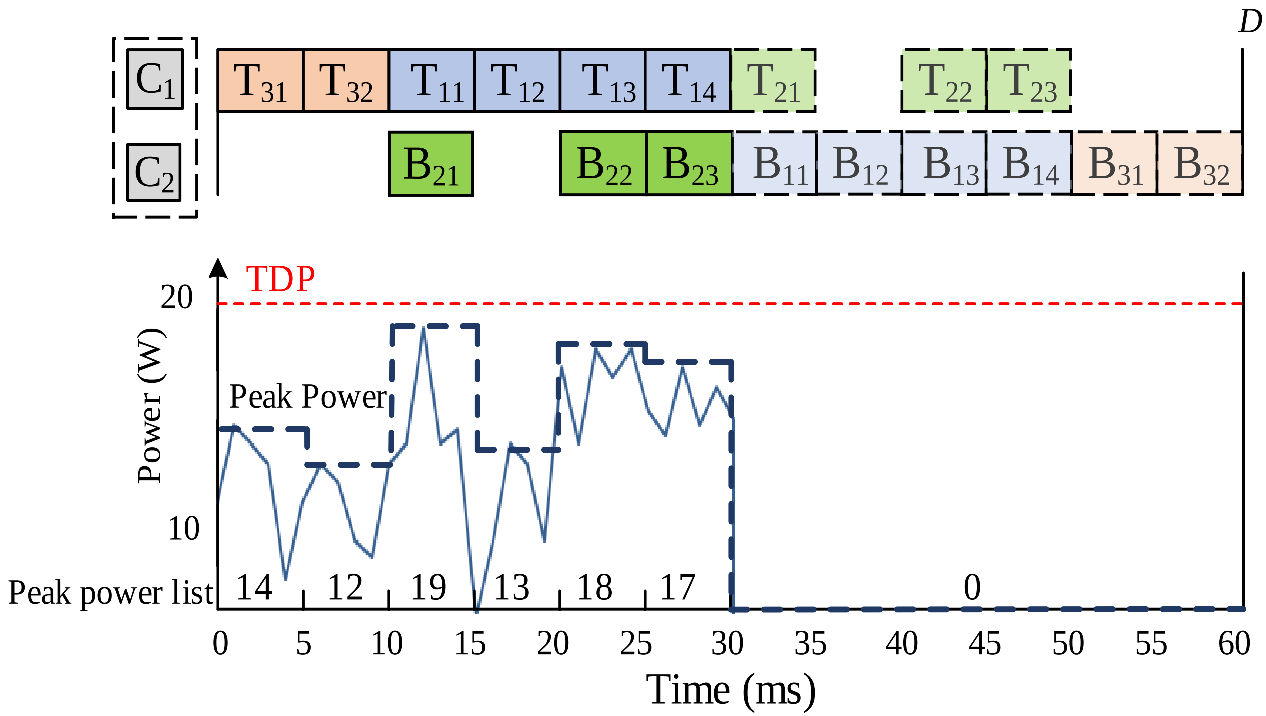
- Proposal: The RAPPM scheduling algorithm
  - The Maximum-Peak-Power-First policy on primary cores
  - The Maximum-Peak-Power-Last policy on spare cores



The RAPPM scheduling on a core pair

Online Phase

- DPM technique
  - Acceptance test



Advantage of fault-free execution

Summary

- Periodic scheduling
  - Rate monotonic scheduling
  - EDF
  - Dependent and sporadic tasks (briefly)
- Power-aware scheduling