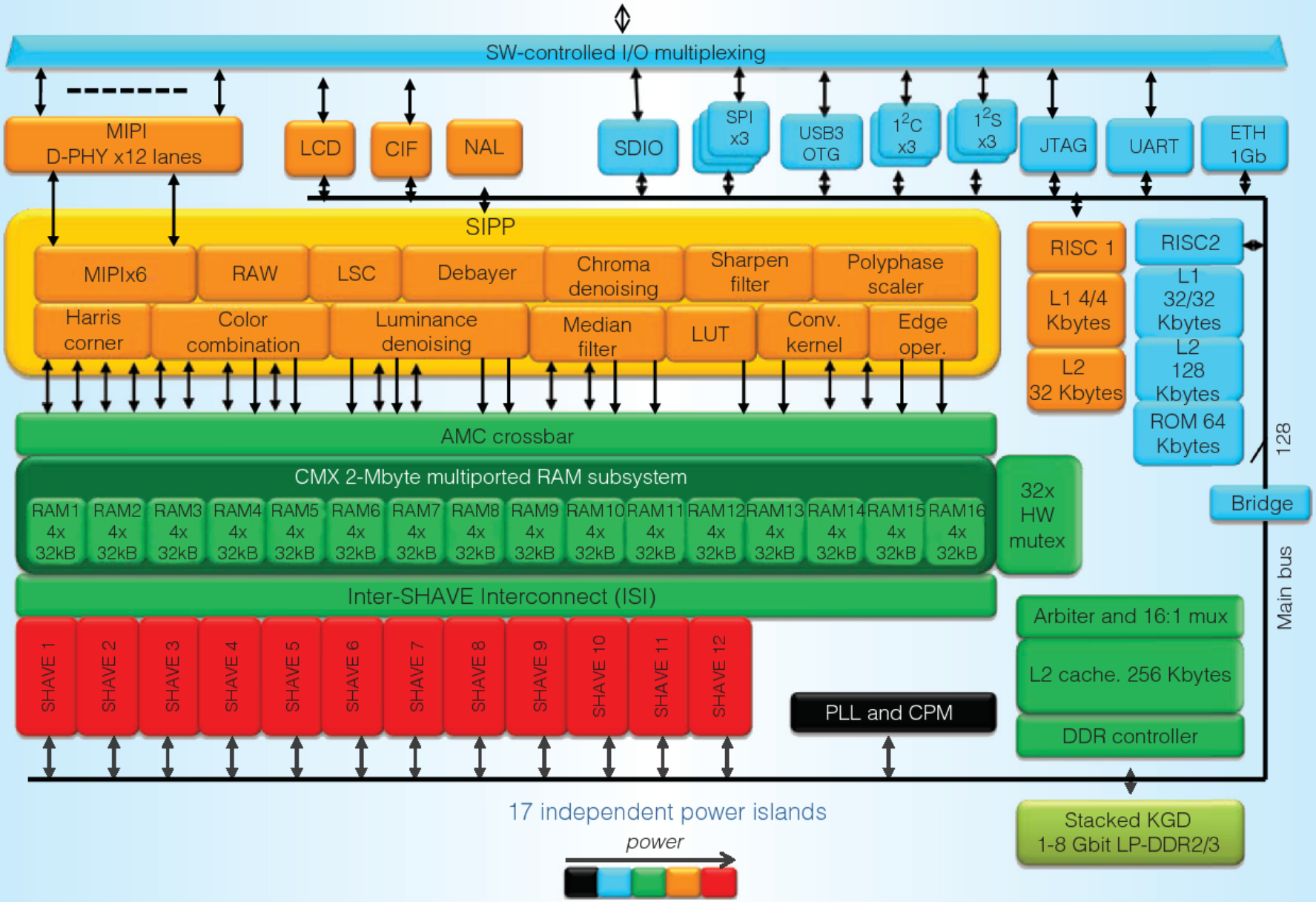
Accelerator Example

- An Embedded System Includes:
  - CPU-usually for management
  - Array Architecture for Speed-Up
  - Bus (AHB usually)
  - RAM
  - Various throughput/protocol ports

myriad 2 vpu: 1 trillion operations per second per watt



Myriad X

- **Programmable 128-bit VLIW Vector Processors**: Run multiple imaging and vision application pipelines simultaneously with the flexibility of 16 vector processors optimized for computer vision workloads.

Myriad X

- **Increased Configurable (Mobile Industry Processor IF) MIPI Lanes**: Connect up to 8 HD resolution RGB cameras directly to Myriad X with its 16 MIPI lanes included in its rich set of interfaces, to support up to 700 million pixels per second of image signal processing throughput.

Myriad X

- **2.5 MB of Homogenous On-Chip Memory**: The centralized on-chip memory architecture allows for up to 450 GB per second of internal bandwidth, minimizing latency and reducing power consumption by minimizing off-chip data transfer.

Embedded Systems – The System Software side

- Real-time scheduling
- Real-time Operating Systems

Real-Time Scheduling

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

Def.: A schedule of G is a mapping

$$V \rightarrow T$$

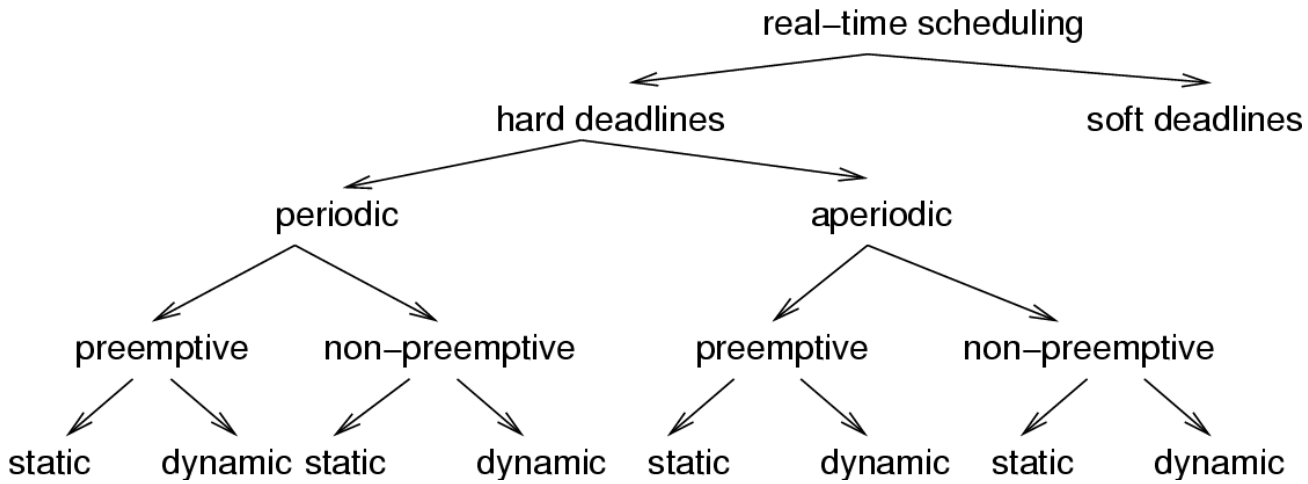
of a set of tasks V to start times from domain T.

Typically, schedules have to respect a number of constraints, such as *resource constraints* and *dependency constraints*, as well as *deadlines*.

Scheduling is the process of finding such a mapping.

During the design of embedded systems, scheduling has to be performed several times

RT scheduling methods (1)

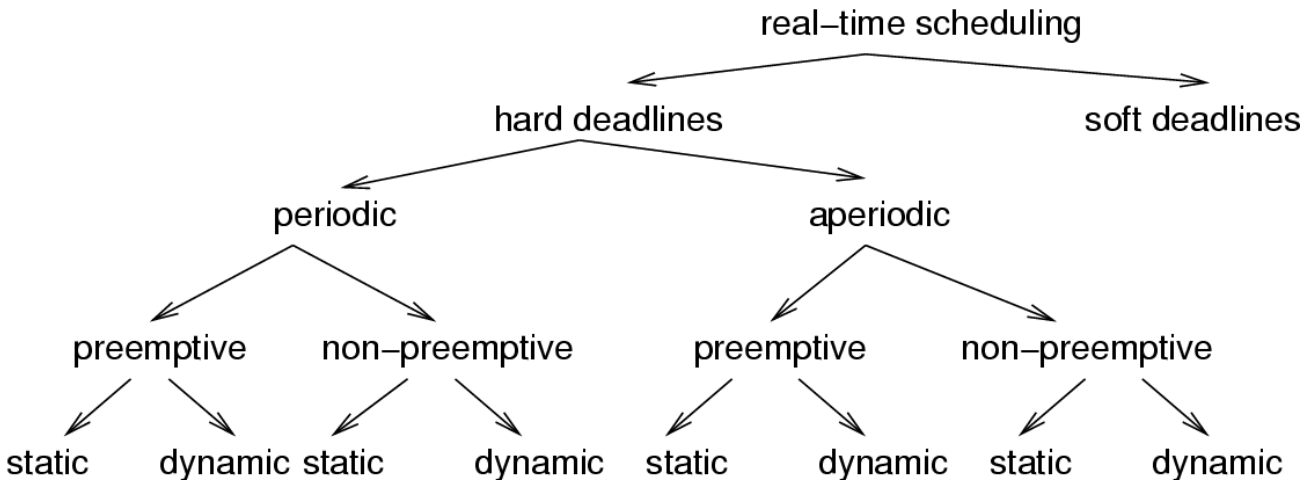


Def.: A time-constraint (deadline) is called **hard** if not meeting that constraint could result in a catastrophe [Kopetz, 1997]. All other time constraints are called **soft**.

Def.: Tasks which must be executed once every p time units are called **periodic** with period p. 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**.

RT scheduling methods (2)



Non-preemptive scheduling: tasks are executed until they are done.

Preemptive schedulers have to be used if some tasks have long execution times or if the response time for external events is required to be short.

Static scheduling: Processor allocation decisions are made at design time. Dispatcher allocates processor when interrupted by a timer. The timer is controlled by a table generated at design time.

Dynamic scheduling: Processor allocation decisions done at run-time. Task priorities can be assigned off- or on-line. Most RTOS use dynamic scheduling.

RT scheduling methods (3)

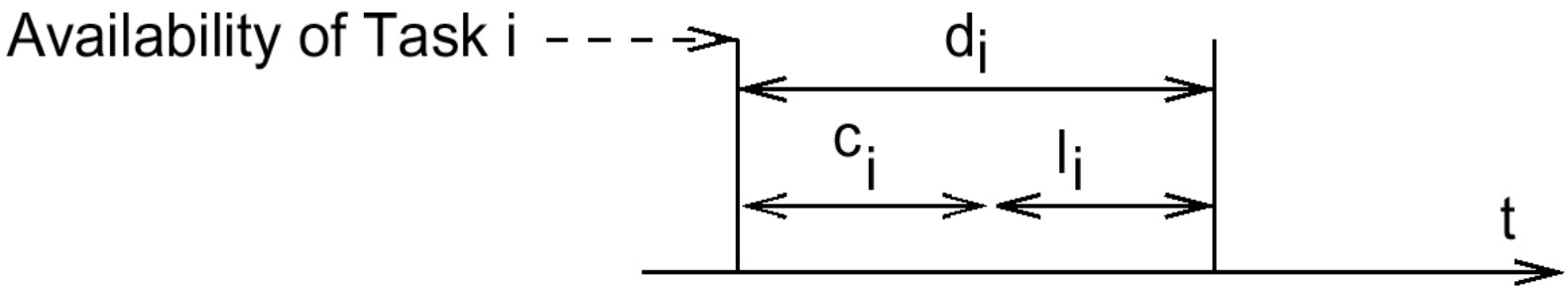
- **Online and offline scheduling:**
  Online: at run-time, based on the information about the tasks arrived so far.
  Offline: taking into account a priori knowledge about arrival times, execution times, and deadlines.
- **Mono- and multi-processor scheduling:**
  Simple scheduling algorithms handle single processors.
  More complex algorithms handle multiple processors.
  algorithms for homogeneous multi-processor systems
- **Centralized and distributed scheduling:** Multiprocessor scheduling with the scheduler executed either locally on one, or on several processors

Schedulability

A set of tasks is said to be ***schedulable*** under a given set of constraints, if a schedule exists for that set of tasks and constraints.

- **Exact tests** are *NP-hard* in many situations.
- **Sufficiency tests**: sufficient conditions for guaranteeing a schedule are checked. Small (hopefully) probability of indicating that no schedule exists even though one exists.
- **Necessary tests**: checking necessary conditions. Can be used to show that no schedule exists. There may be cases in which no schedule exists and we may still be unable to prove this.

Aperiodic scheduling (1)



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 T_i has to finish execution.

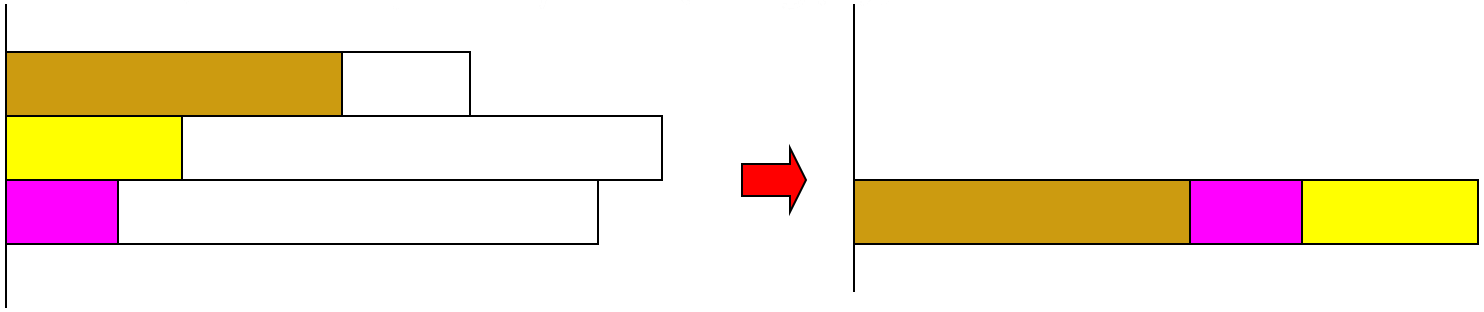
l_i be the laxity or slack, defined as $l_i = d_i - c_i$

Aperiodic scheduling (2)

- **Cost function:** Different algorithms aim at minimizing different cost functions.
- **Def.: Maximum lateness** is the difference between the completion time and the deadline, maximized over all tasks. Maximum lateness is negative if all tasks complete before their deadline.
- **Objective:** Minimize the Maximum lateness

Uniprocessor, **same** task arrival time

- Preemption is useless.
- **Earliest Due Date** (EDD): Based on Jackson's rule:
  Given a set of *n* independent tasks, any algorithm that executes the tasks in order of nondecreasing deadlines is optimal **with respect to minimizing the maximum lateness**. Proof: See [Buttazzo, 2002]
- EDD requires all tasks to be sorted by their deadlines.
  Hence, its complexity is *O(n log(n))*.



Uniprocessor, **different** task arrival times

- Different arrival times: **Preemption may potentially reduce 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.

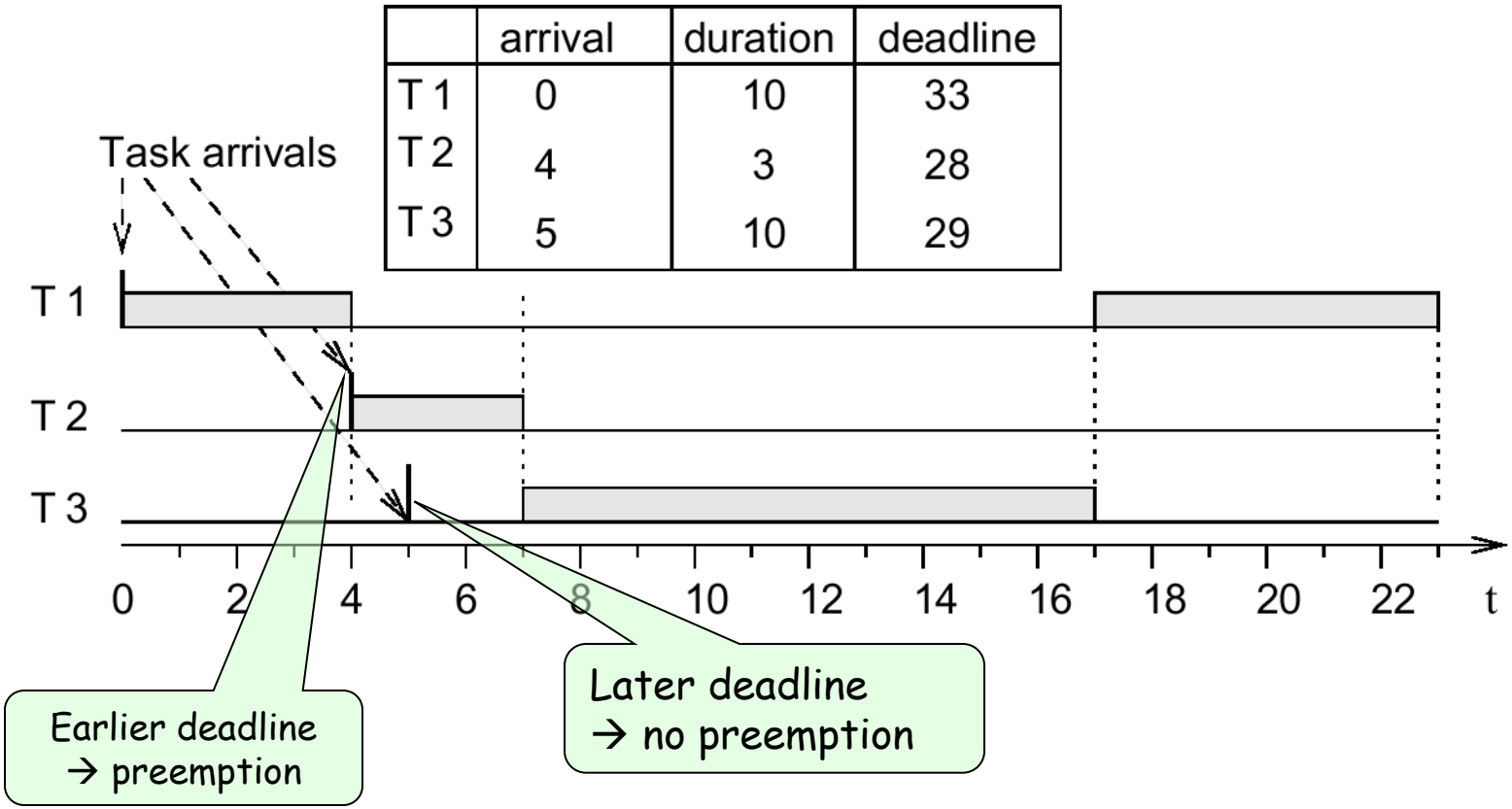
- **Earliest deadline first** (EDF) algorithm:

Each time a new ready task arrives, it is inserted into a queue of ready tasks, sorted by their deadlines. If a newly arrived task is inserted at the head of the queue, the currently executing task is preempted.

- Dynamic scheduling, priorities fixed off-line based on known deadlines.
- If sorted linked lists are used, the complexity is in $O(n^2)$

Earliest Deadline First (EDF)

	arrival	duration	deadline
T 1	0	10	33
T 2	4	3	28
T 3	5	10	29



Least Slack Time First scheduling

Priorities = decreasing function of the laxity i.e. the less the laxity, the higher the priority; dynamically changing priority; preemptive.

Worse than EDF, because it requires knowledge of task execution times.

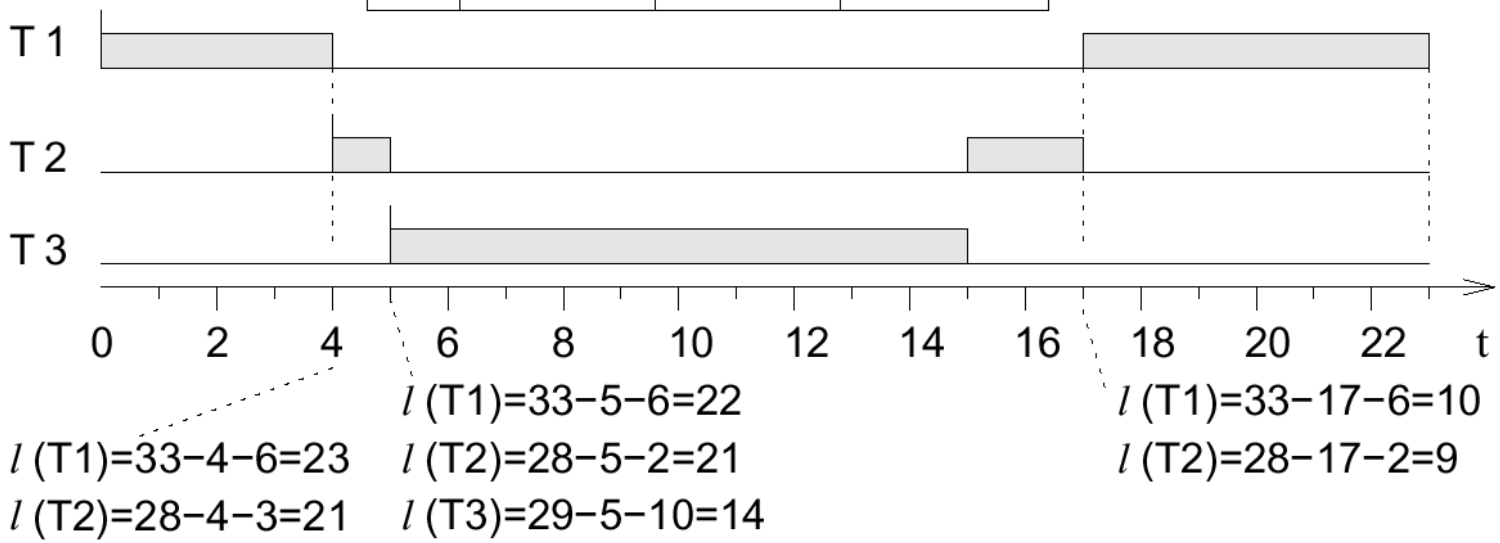
Requires calling the scheduler periodically, and to recompute the laxity. Overhead for many calls of the scheduler and many context switches.

Detects missed deadlines early

Optimal for uniprocessors, in the sense that it will find a schedule if one exists

Least Slack Time First (LST)

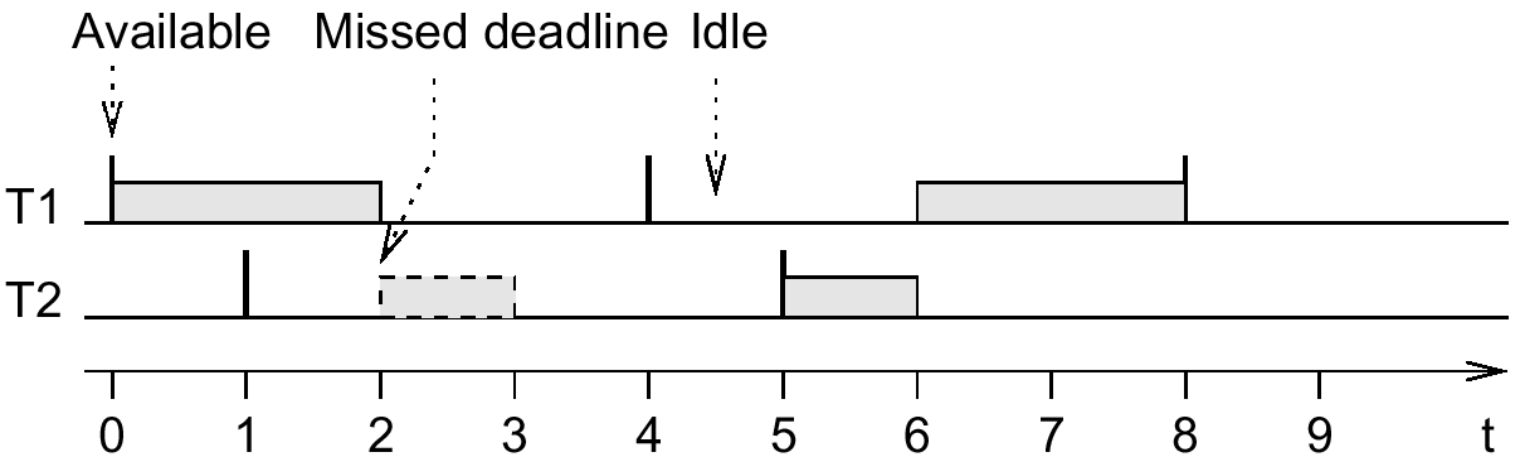
	arrival	duration	deadline
T 1	0	10	33
T 2	4	3	28
T 3	5	10	29



Scheduling with no preemption

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.



T1: periodic, $c_1 = 2$, $p_1 = 4$, $d_1 = 4$

T2: occasionally available at times $4*n+1$, $c_2 = 1$, $d_2 = 1$

T1 has to start at t=0

→ deadline missed, but schedule is possible (start T2 first)

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

Scheduling without preemption

- If preemption is not allowed → optimal schedules may have to leave processor idle to finish tasks with early deadlines arriving late.

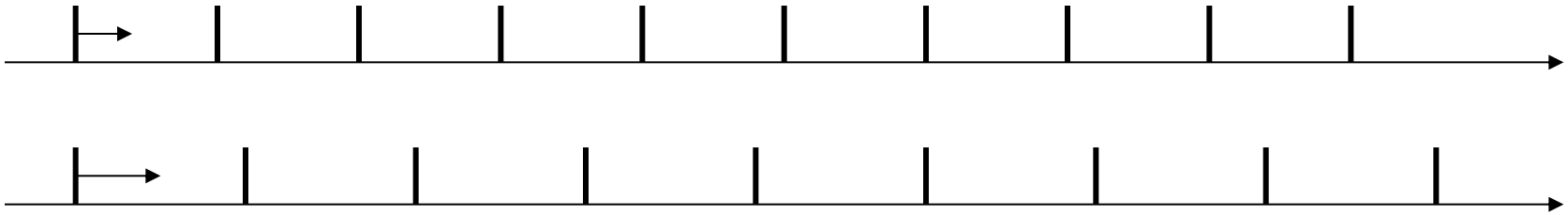
→i.e. knowledge about the future tasks arrival is needed for optimal scheduling

→No online algorithm can decide whether or not to keep idling.

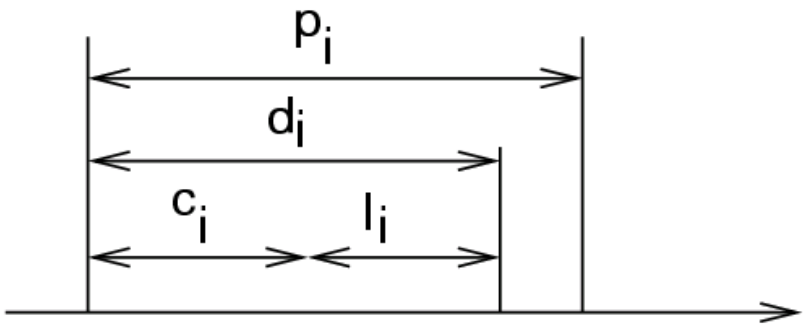
- EDF is optimal among all scheduling algorithms that are not keeping the processor idle at certain times.

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

Periodic Scheduling



A scheduler is considered **optimal** if and only if it will find a valid schedule if one exists.



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,
l_i be the **laxity** or **slack**, defined as $l_i = d_i - c_i$

Schedulability

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

Accumulated utilization

Necessary condition for schedulability:
(m = number of available processors):

$$\mu \leq m$$

Rate Monotonic Scheduling (RMS)

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

Assumptions:

1. All tasks that have hard deadlines are periodic.
2. All tasks are independent.
3. $d_i=p_i$, for all tasks.
4. c_i is constant and is known for all tasks.
5. The time required for context switching is negligible.
6. For a single processor and for n tasks, the following equation holds for the accumulated utilization μ

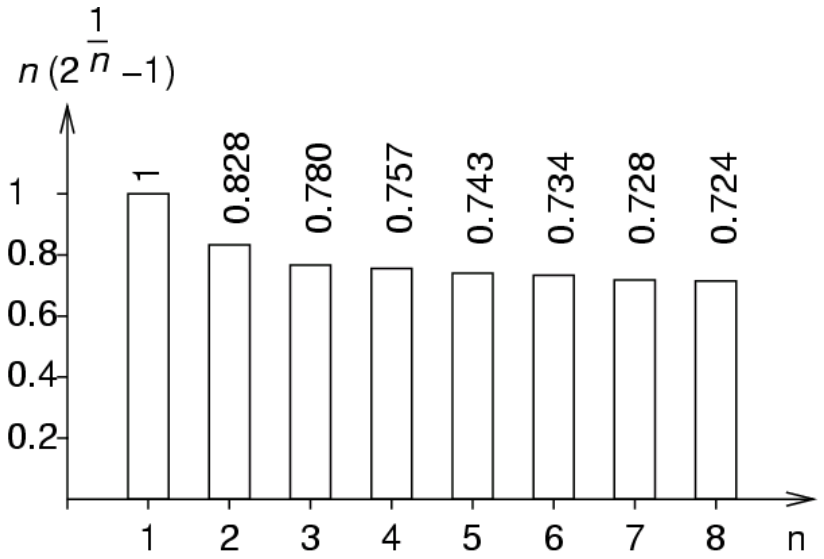
$$\mu = \sum_{i=1}^{n} \frac{c_i}{p_i} \leq n(2^{1/n} - 1)$$

The RMS Policy

The priority of a task is a monotonically decreasing function of its period. I.e. the lower the period the higher the priority

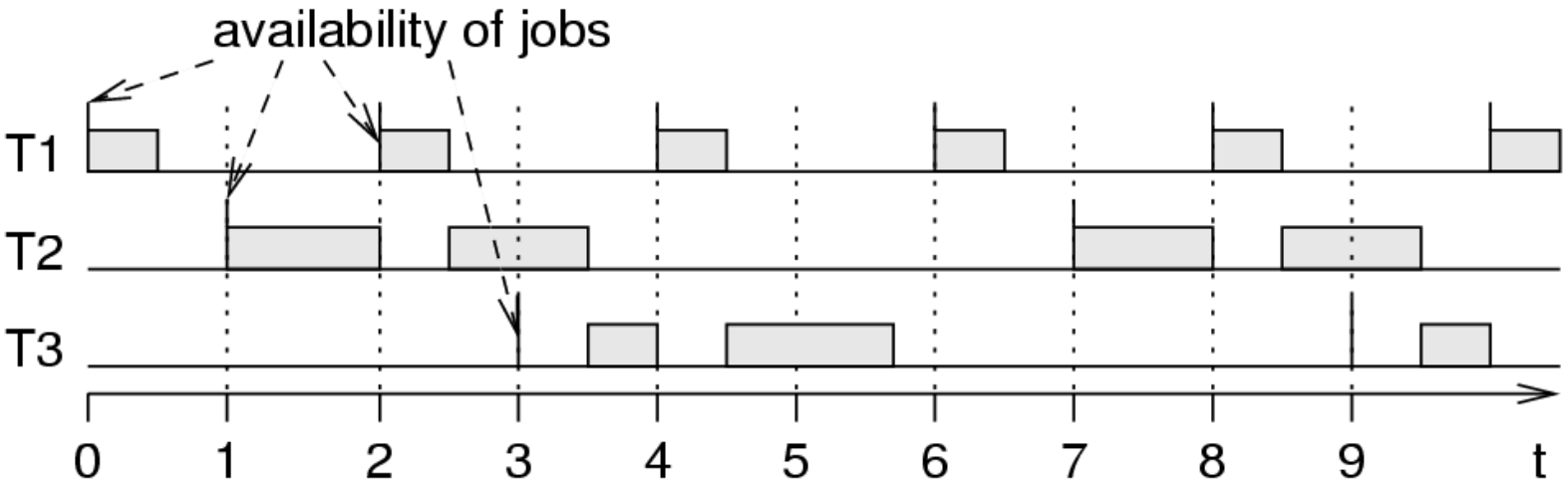
At any time, a highest priority task, among all those tasks that are ready for execution, is allocated.

If all assumptions are met, schedulability is guaranteed.



Maximum utilization as a function of the number of tasks

RMS example – deadline not missed

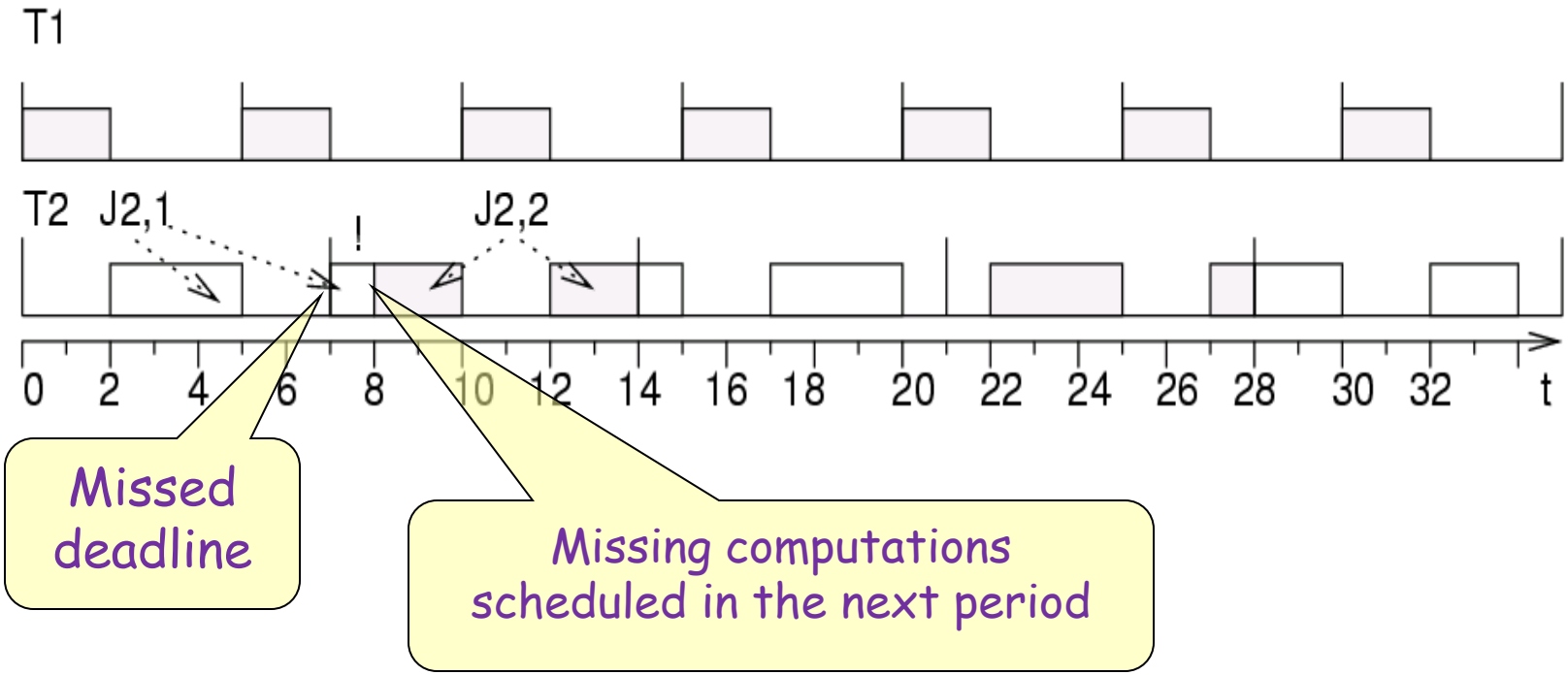


T1: period =2, execution time = 0.5, highest priority

T2: period =6, execution time = 2, same priority as T3

T3: period = 6, execution time = 1.75, same priority as T2

RM scheduling – deadline missed



T1: period = 5, execution time = 2
T2: period = 7, execution time = 4
$\mu = 2/5+4/7 = 34/35 \approx 0.97 > 2(2^{1/2}-1) \approx 0.828$, violation!

RMS properties - summary

- A huge number of variations of RM scheduling exists. In the context of RM scheduling, many formal proofs exist.
- The idle capacity is not required if the period of all tasks is a multiple of the period of the highest priority task, that is, schedulability is then also guaranteed if $\mu \leq 1$.
- RM scheduling is based on **static** priorities. This allows RM scheduling to be used in standard O/S, such as Windows NT.

EDF periodic scheduling

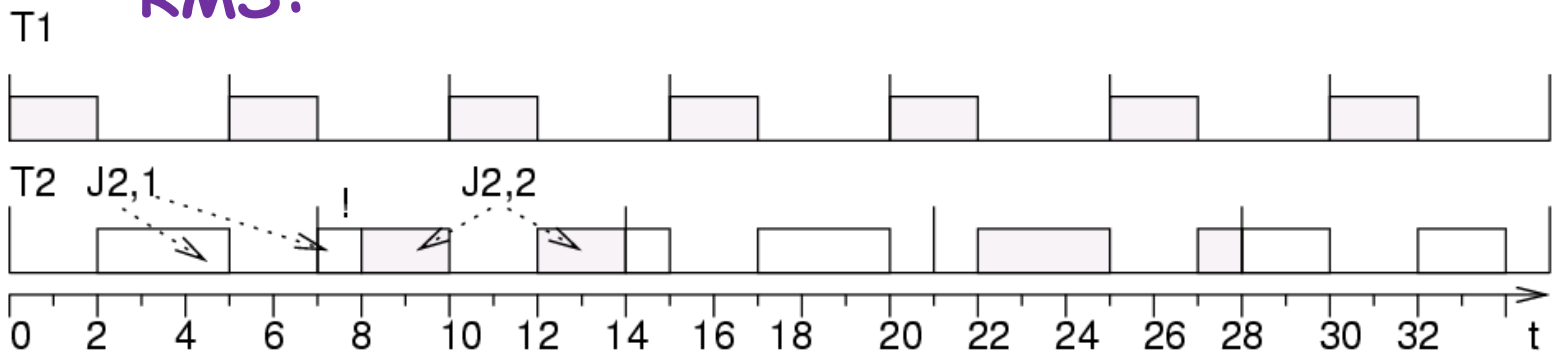
EDF can also be applied to periodic scheduling.

EDF is optimal for every period

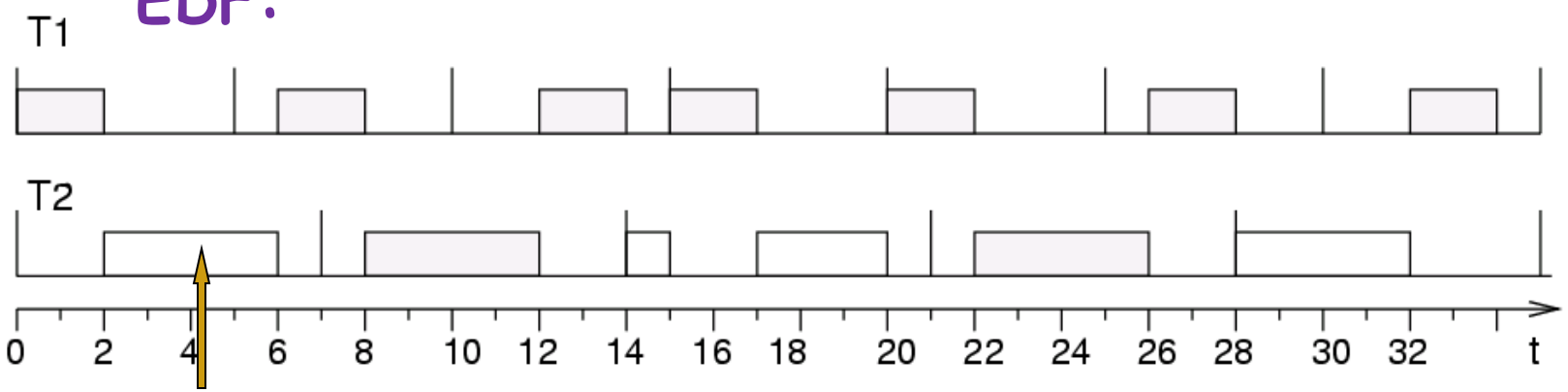
- ➔ optimal for periodic scheduling
- ➔ EDF must be able to schedule the example in which RMS failed.

RMS/EDF comparison

RMS:



EDF:



T2 not preempted, due to its earlier deadline.

Scheduling 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 number of decisions needs to be made at run-time.

Handling 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.