Reuse of standard software components

Knowledge from previous designs to be made available in the form of **intellectual property** (IP, for SW & HW).

- Operating systems
- Middleware
- Real-time data bases
- Standard software (MPEG-x, GSM-kernel, …)

Includes standard approaches for scheduling (requires knowledge about execution times).

Worst case execution times

Def.: The **worst case execution time** (WCET) is an **upper bound** on the execution times of tasks.

The term is not ideal, since a program requiring the WCET for its execution does not have to exist (WCET is a **bound**).

Complexity:

- in the general case: undecidable if a bound exists.
- for restricted programs: simple for „old“ architectures, very complex for new architectures with pipelines, caches, interrupts, virtual memory, etc.

Approaches:

- for hardware: typically requires hardware synthesis
- for software: requires availability of machine programs; complex analysis (see, e.g., www.absint.de)

Average execution times

- **Estimated cost and performance values:** Difficult to generate sufficiently precise estimates; Balance between run-time and precision
- **Accurate cost and performance values:** Can be done with normal tools (such as compilers). As precise as the input data is.

Real-time scheduling

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

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

$$V \rightarrow T$$

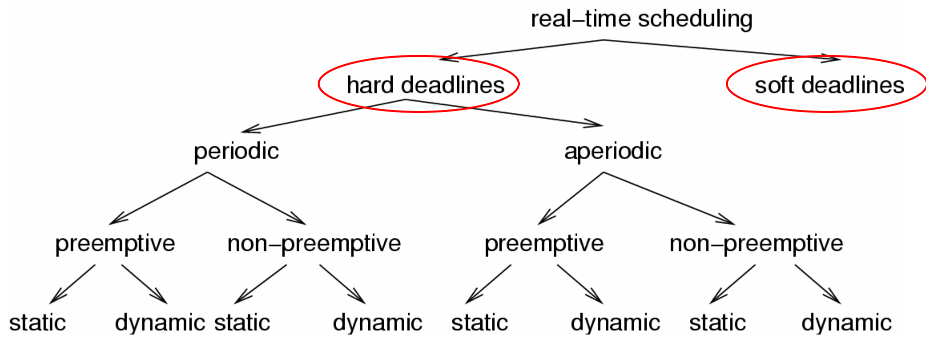
of a set of tasks V to start times from domain T.

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

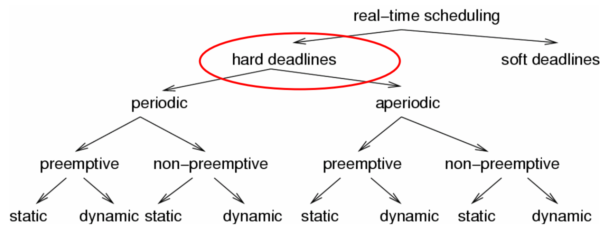
Scheduling is the process of finding such a mapping.

During the design of embedded systems, scheduling has to be performed several times (early rough scheduling as well as late precise scheduling).

Classification of scheduling algorithms



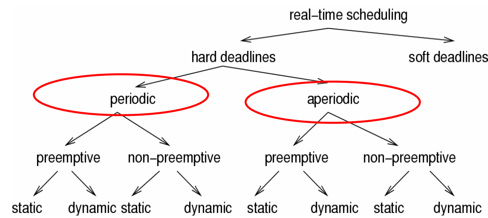
Hard and soft deadlines



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

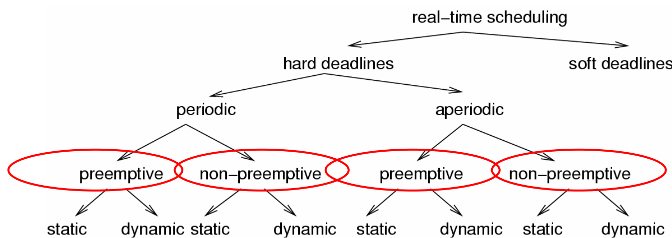


Def.: Tasks which must be executed once every *p* units of time are called **periodic** tasks. *p* is called their period. Each execution of a periodic task is called a **job**.

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

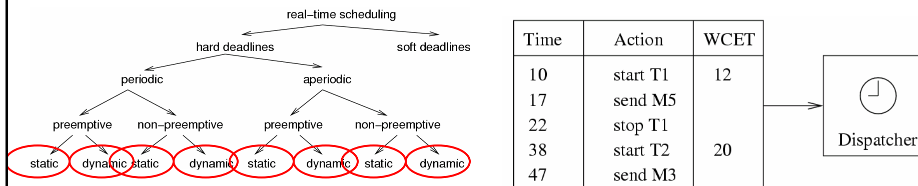


Preemptive and non-preemptive scheduling: Non-preemptive schedulers are based on the assumption that tasks are executed until they are done. As a result the response time for external events may be quite long if some tasks have a large execution time. Preemptive schedulers have to be used if some tasks have long execution times or if the response time for external events is required to be short.

Static and dynamic scheduling

Dynamic scheduling: Processor allocation decisions (scheduling) done at run-time.

Static scheduling: Processor allocation decisions (scheduling) done at design-time. Dispatcher allocates processor when interrupted by a timer. The timer is 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

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]

… pre-run-time scheduling is often the only practical means of providing predictability in a complex system. ([Xu, Parnas], as cited by Kopetz).

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

Centralized and distributed scheduling: Multiprocessor scheduling either locally on 1 or on several processors

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 hardware accelerators as a special case).

Online- and offline scheduling:

- Online: scheduling at run-time, based on the information about the tasks arrived so far.
- Offline: scheduling taking a priori knowledge about arrival times, execution times, and deadlines into account.

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.

Sufficient 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.

Cost functions

Cost function: Different algorithms aim at minimizing different functions.

Def.: Maximum lateness is defined as the difference between the completion time and the deadline, maximized over all tasks. Maximum lateness is negative if all tasks complete before their deadline.

Simple tasks

Tasks without any interprocess communication are called **simple tasks** (S-tasks).

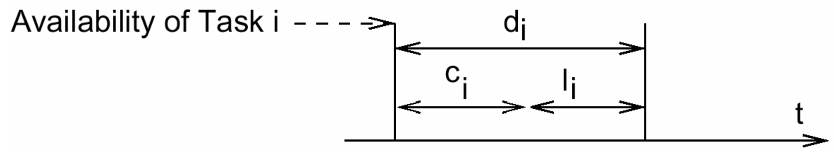
S-tasks can be in one out of two states: ready or running.

The API of a TT-OS supporting S-tasks is quite simple: *The application program interface (API) of an S-task in a TT system consists of three data structures and two operating system calls. ... The system calls are TERMINATE TASK and ERROR. The TERMINATE TASK system call is executed whenever the task has reached its termination point. In case of an error that cannot be handled within the application task, the task terminates its operation with the ERROR system call.* [Kopetz, 1997].

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 $l_i = d_i - c_i$

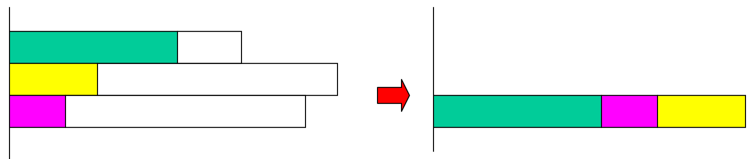


Uniprocessor with equal arrival times

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))*.



Earliest Deadline First (EDF) - Algorithm -

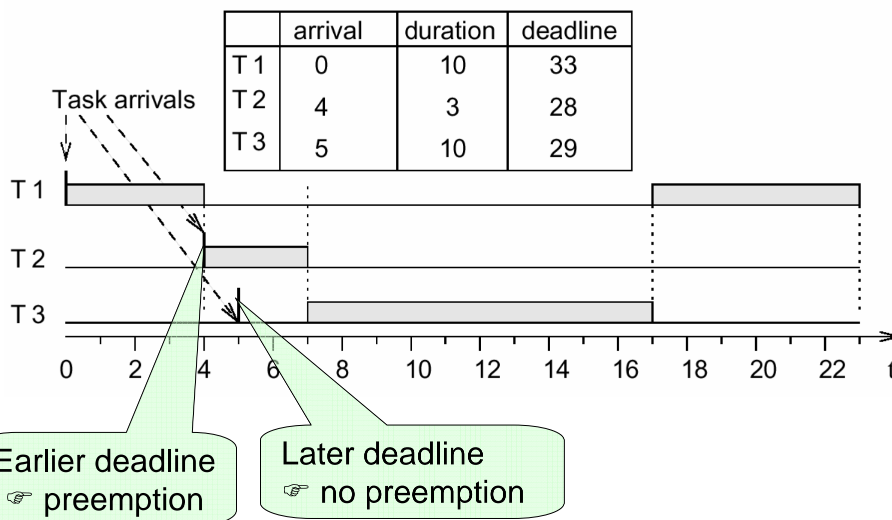
Different arrival times: Preemption potentially reduces lateness.

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: 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.

If sorted lists are used, the complexity is $O(n^2)$ (less with bucket arrays).

Earliest Deadline First (EDF) - Example -

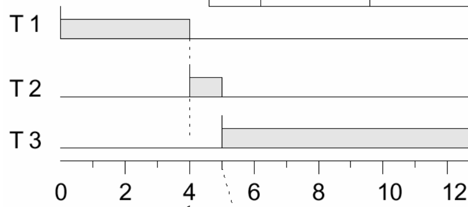


	arrival	duration	deadline
T 1	0	10	33
T 2	4	3	28
T 3	5	10	29

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

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

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



l (T1)=33−5−6=22

l (T1)=33−4−6=23 l (T2)=28−5−2=21

l (T2)=28−4−3=21 l (T3)=29−5−10=14

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.

Properties

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

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

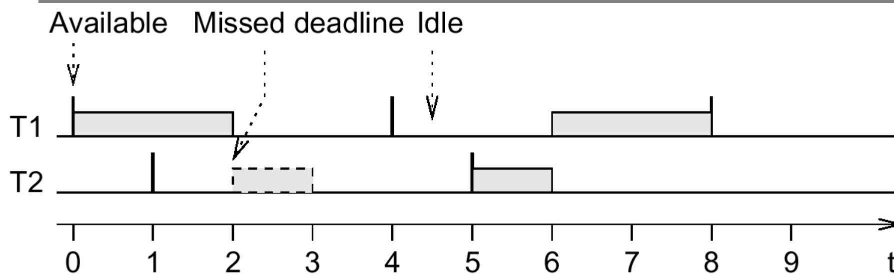
T1: periodic, $c_1 = 2$, $p_1 = 4$, $d_1 = 4$

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

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

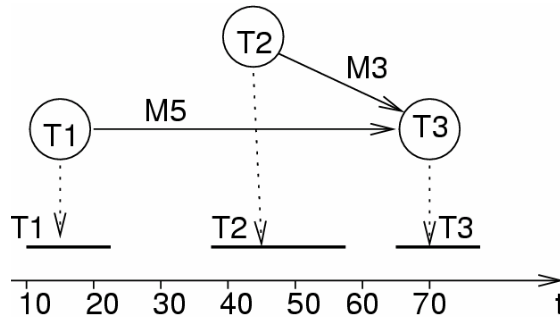
☞No online algorithm can decide whether or not to keep idle.

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.

Scheduling with precedence constraints

Task graph and possible schedule:



Schedule can be stored in table.

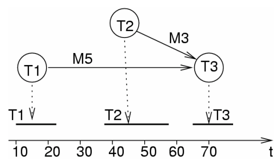
Simultaneous Arrival Times: The Latest Deadline First (LDF) Algorithm

LDF [Lawler, 1973]: Generation of total order compatible with the partial order described by the task graph (LDF performs a topological sort).

LDF reads the task graph and inserts tasks with no successors 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.



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 uni-processor systems.

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