# Chapter 10
# Real-Time Scheduling

**Overview** Many thousands of research papers have been written about how to schedule a set of tasks in a system with a limited amount of resources such that all tasks will meet their deadlines. This chapter tries to summarize some important results of scheduling research that are relevant to the designer of real-time systems. The chapter starts by introducing the notion of a schedulability test to determine whether a given task set is schedulable or not. It distinguishes between a sufficient, an exact, and a necessary schedulability test. A scheduling algorithm is *optimal* if it will find a schedule whenever there is a solution. The adversary argument shows that generally it is not possible to design an optimal on-line scheduling algorithm. A prerequisite for the application of any scheduling technique is knowledge about the worst-case execution time (WCET) of all time-critical tasks. Section 10.2 presents techniques to estimate the WCET of simple tasks and complex tasks. Modern processors with pipelines and caches make it difficult to arrive at tight bounds for the WCET. *Anytime algorithms* that contain a root segment that provides a result of sufficient (but low) quality and an optional periodic segment that improves on the quality of the previous result point to a way out of this dilemma. They use the interval between the actual execution time of the root segment of a concrete task execution and the deadline, i.e., the worst execution time of the root segment, to improve the quality of the result. Section 10.3 covers the topic of static scheduling. The concept of the schedule period is introduced and an example of a simple search tree that covers a schedule period is given. A heuristic algorithm has to examine the search tree to find a feasible schedule. If it finds one, the solution can be considered a constructive schedulability test. Section 10.4 elaborates on dynamic scheduling. It starts by looking at the problem of scheduling a set of independent tasks by the rate-monotonic algorithm. Next, the problem of scheduling a set of dependent tasks is investigated. The priority-ceiling protocol is introduced and a schedulability test for the priority ceiling protocol is sketched. Finally, the scheduling problem in distributed systems is touched and some ideas about alternative scheduling strategies such as feedback scheduling are given.

## 10.1 The Scheduling Problem

A hard real-time system must execute a set of concurrent real-time tasks in such a way that all time-critical tasks meet their specified deadlines. Every task needs computational, data, and other resources (e.g. input/output devices) to proceed. The scheduling problem is concerned with the allocation of these resources to satisfy all timing requirements.

### *10.1.1 Classification of Scheduling Algorithms*

The following diagram (Fig. 10.1) presents a taxonomy of real-time scheduling algorithms [Che87].

*Static Versus Dynamic Scheduling*. A scheduler is called static (or pre-run-time) if it makes its scheduling decisions at compile time. It generates a dispatching table for the run-time dispatcher off-line. For this purpose it needs complete prior knowledge about the task-set characteristics, e.g., maximum execution times, precedence constraints, mutual exclusion constraints, and deadlines. The dispatching table (see Fig. 9.2) contains all information the dispatcher needs at run time to decide at every point of the sparse time-base which task is to be scheduled next. The run-time overhead of the dispatcher is small. The system behavior is deterministic.

A scheduler is called *dynamic* (or *on-line*) if it makes its scheduling decisions at run time, selecting one out of the current set of *ready* tasks. Dynamic schedulers are flexible and adapt to an evolving task scenario. They consider only the *current* task requests. The run-time effort involved in finding a schedule can be substantial. In general, the system behavior is non-deterministic.

*Non-preemptive and Preemptive Scheduling*. In non-preemptive scheduling, the currently executing task will not be interrupted until it decides on its own to release the allocated resources. Non-preemptive scheduling is reasonable in a task scenario where many short tasks (compared to the time it takes for a context switch) must be executed. In preemptive scheduling, the currently executing task may be preempted, i. e., interrupted, if a more urgent task requests service.

*Centralized Versus Distributed Scheduling*. In a dynamic distributed real-time system, it is possible to make all scheduling decisions at one central site or to

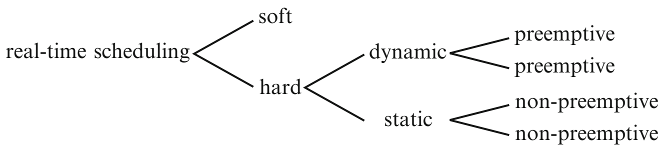


**Fig. 10.1** Taxonomy of real-time scheduling algorithms

develop cooperative distributed algorithms for the solution of the scheduling problem. The central scheduler in a distributed system is a critical point of failure. Because it requires up-to-date information on the load situations of all nodes, it can also contribute to a communication bottleneck.

### *10.1.2 Schedulability Test*

A test that determines whether a set of ready tasks can be scheduled such that each task meets its deadline is called a *schedulability test*. We distinguish between *exact*, *necessary*, and *sufficient* schedulability tests (Fig. 10.2).

A scheduler is called *optimal* if it will always find a feasible schedule whenever it exists. Alternatively, a scheduler is called *optimal*, if it can find a schedule whenever a clairvoyant scheduler, i.e., a scheduler with complete knowledge of the future request times, can find a schedule. Garey and Johnson [Gar75] have shown that in nearly all cases of task dependency, even if there is only one common resource, the complexity of an exact schedulability test algorithm belongs to the class of NP-complete problems and is thus computationally intractable. *Sufficient schedulability test* algorithms can be simpler at the expense of giving a negative result for some task sets that are, in fact, schedulable. A task set is definitely not schedulable if a *necessary schedulability* test gives a negative result. If a necessary schedulability test gives a positive result, there is still a probability that the task set may not be schedulable. The *task request time* is the instant when a request for a task execution is made. Based on the request times, it is useful to distinguish between two different task types: *periodic* and *sporadic* tasks. This distinction is important from the point of view of schedulability.

If we start with an initial request, all future request times of a periodic task are known a priori by adding multiples of the known period to the initial request time. Let us assume that there is a task set $\{T_i\}$ of periodic tasks with periods $p_i$, deadline interval $d_i$, and execution time $c_i$. The deadline interval is the duration between the deadline of a task and the task request instant, i.e., the instant when a task becomes ready for execution. We call the difference $d_i - c_i$ the *laxity* $l_i$ of a task. It is sufficient to examine schedules of length of the least common multiples of the periods of these tasks, the *schedule period*, to determine schedulability.

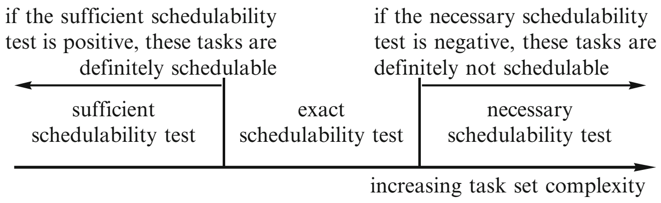

if the sufficient schedulability test is positive, these tasks are definitely schedulable
if the necessary schedulability test is negative, these tasks are definitely not schedulable
sufficient schedulability test
exact schedulability test
necessary schedulability test
increasing task set complexity

**Fig. 10.2** Necessary and sufficient schedulability test

A necessary schedulability test for a set of periodic tasks states that the sum of the utilization factors

$$\mu = \sum \mathrm{c_i/p_i} \leqslant \mathrm{n},$$

must be less or equal to *n*, where *n* is the number of available processors. This is evident because the utilization factor of task $T_i$, $\mu_i$, denotes the percentage of time task $T_i$ requires service from a processor.

The request times of *sporadic* tasks are not known a priori. To be schedulable, there must be a minimum interval between any two request times of sporadic tasks. Otherwise, the necessary schedulability test introduced above will fail. If there is no constraint on the request times of task activations, the task is called an *aperiodic* task.

### 10.1.3 The Adversary Argument

Let us assume that a real-time computer system contains a dynamic scheduler with full knowledge of the past but without any knowledge about future request times of tasks. Schedulability of the current task set may depend on when a sporadic task will request service in the future.

The *adversary argument* [Mok93, p. 41] states that, in general, it is not possible to construct an optimal totally on-line dynamic scheduler if there are mutual exclusion constraints between a periodic and a sporadic task. The proof of the adversary argument is relatively simple.

Consider two mutually exclusive tasks, task *T*1 is periodic and the other task *T*2 is sporadic, with the parameters given in Fig. 10.3. The necessary schedulability test introduced above is satisfied, because

$$\mu = 2/4 + 1/4 = 3/4 \leqslant 1.$$

Whenever the periodic task is executing, an *adversary* requests service for the sporadic task. Due to the mutual exclusion constraint, the sporadic task must wait until the periodic task is finished. Since the sporadic task has a laxity of 0, it will miss its deadline.

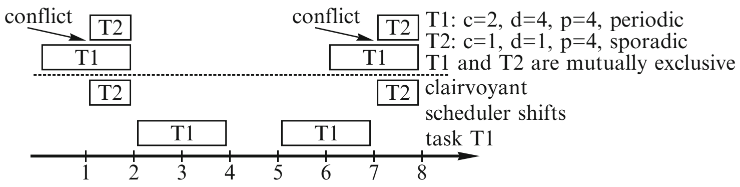

conflict
T2
T1
conflict
T2
T1
T1: c=2, d=4, p=4, periodic
T2: c=1, d=1, p=4, sporadic
T1 and T2 are mutually exclusive
T2
T2
clairvoyant scheduler shifts task T1
T1
T1
1
2
3
4
5
6
7
8

**Fig. 10.3** The adversary argument

The clairvoyant scheduler knows all the future request times of the sporadic tasks and at first schedules the sporadic task and thereafter the periodic task in the gap between two sporadic task activations (Fig. 10.3).

The adversary argument demonstrates the importance of information about the future behavior of tasks for solving the scheduling problem. If the on-line scheduler does not have any further knowledge about the request times of the sporadic task, the dynamic scheduling problem is not solvable, although the processor capacity is more than sufficient for the given task scenario. The design of predictable hard real-time systems is simplified if *regularity assumptions* about the future scheduling requests can be made. This is the case in cyclic systems that restrain the points in time at which external requests are recognized by the computing system.

## 10.2 Worst-Case Execution Time

A deadline for completing an RT transaction can only be guaranteed if the worst-case execution times (WCET) of all application tasks and communication actions that are part of the RT-transaction are known a priori. The WCET of a task is a guaranteed upper bound for the time between task activation and task termination. It must be valid for all possible input data and execution scenarios of the task and should be a tight bound.

In addition to the knowledge about the WCET of the application tasks, we must find an upper bound for the delays caused by the administrative services of the operating system, the worst-case administrative overhead (WCAO). The WCAO includes all administrative delays that affect an application task but are not under the direct control of the application task (e.g., those caused by context switches, scheduling, cache reloading because of task preemption by interrupts or blocking, and direct memory access).

This section starts with an analysis of the WCET of a non-preemptive simple task. We then proceed to investigate the WCET of a preemptive simple task before looking at the WCET of complex tasks and, finally, we discuss the state of the art regarding the timing analysis of real-time programs.

### *10.2.1 WCET of Simple Tasks*

The simplest task we can envision is a single sequential S-task that runs on dedicated hardware without preemption and without requiring any operating system services. The WCET of such a task depends on:

1. The source code of the task
2. The properties of the object code generated by the compiler
3. The characteristics of the target hardware

In this section, we investigate the analytical construction of a tight worst-case execution time bound of such a simple task on hardware, where the execution time of an instruction is context independent.

*Source Code Analysis.* The first problem concerns the calculation of the WCET of a program written in a higher-level language, under the assumption that the maximum execution times of the basic language constructs are known and context independent. In general, the problem of determining the WCET of an arbitrary sequential program is unsolvable and is equivalent to the halting problem for Turing machines. Consider, for example, the simple statement that controls the entry to a loop:

```
S: while(exp)
     do loop;
```

It is not possible to determine a priori after how many iterations, if at all, the Boolean expression *exp* will evaluate to the value FALSE and when statement S will terminate. For the determination of the WCET to be a tractable problem there are a number of constraints that must be met by a the program [Pus89]:

1. Absence of unbounded control statements at the beginning of a loop
2. Absence of recursive function calls
3. Absence of dynamic data structures

The WCET analysis concerns only the temporal properties of a program. The temporal characteristics of a program can be abstracted into a WCET bound for every program statement using the known WCET bound of the basic language constructs. For example, the WCET bound of a conditional statement

```
S: if(exp)
     then S1
     else S2;
```

can be abstracted as

$$T(S) = max\,[T(exp) + T(S_1),\ T(exp) + T(S_2)]$$

where $T(S)$ is the maximum execution time of statement $S$, with $T(exp)$, $T(S_1)$, and $T(S_2)$ being the WCET bounds of the respective constructs. Such a formula for reasoning about the timing behavior of a program is called a *timing schema* [Sha89].

The WCET analysis of a program which is written in a high-level language must determine which program path, i.e., which sequence of instructions, will be executed in the worst-case scenario. The longest program path is called the *critical path.* Because the number of program paths normally grows exponentially with the program size, the search for the critical path can become intractable if the search is not properly guided and the search space is not reduced by excluding infeasible paths.

*Compiler Analysis.* The next problem concerns the determination of the maximum execution time of the basic language constructs of the source language under the assumption that the maximum execution times of the machine language commands are known and context independent. For this purpose, the code generation strategy of the compiler must be analyzed, and the timing information that is available at the source code level must be mapped into the object code representation of the program such that an object-code timing analysis tool can make use of this information.

*Execution Time Analysis.* The next problem concerns the determination of the worst-case execution time of the commands on the target hardware. If the processor of the target hardware has fixed instruction execution times, the duration of the hardware instructions can be found in the hardware documentation and can be retrieved by an elementary table look-up. Such a simple approach does not work if the target hardware is a modern RISC processor with pipelined execution units and instruction/data caches. While these architectural features result in significant performance improvements, they also introduce a high level of unpredictability. Dependencies among instructions can cause pipeline hazards, and cache misses will lead to a significant delay of the instruction execution. To make things worse, these two effects are not independent. A significant amount of research deals with the execution time analysis on machines with pipelines and caches. The excellent survey article by Wilhelm et al. [Wil08] presents the state of the art of WCET analysis in research and industry and describes many of the tools available for the support of WCET analysis.

*Preemptive S-Tasks.* If a simple task (S task) is preempted by another independent task, e.g., a higher priority task that must service a pending interrupt, the execution time of the S-task under consideration is extended by three terms:

1. The WCET of the interrupting task (task B in Fig. 10.4)
2. The WCET of the operating system required for context switching
3. The time required for reloading the instruction cache and the data cache of the processor whenever the context of the processor is switched

We call the sum of the worst-case delays caused by the context switch (2), and the cache reloading (3) the Worst-Case Administrative Overhead (WCAO) of a task preemption. The WCAO is an unproductive administrative operating system overhead that is avoided if task preemption is forbidden.

The additional delay caused by the preemption of task A by task B is the WCET of the independent task B and the sum of the two WCAOs for the two context switches (shaded area in Fig. 10.4). The times spent in Microarchitecture-1 and Microarchitecture-2 are the delays caused by cache reloading. The Microarchitecture-2 time of the first context switch is part of the WCET of task B, because task B

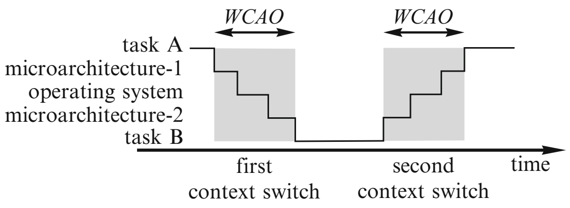


**Fig. 10.4** Worst-case administrative overhead (WCAO) of a task preemption

is assumed to start on an empty cache. The second context switch includes the cache reload time of task A, because in a non-preemptive system, this delay would not occur. In many applications with modern processors, the micro-architecture delays can be the significant terms that determine the cost of task preemption because the WCET of the interrupting task is normally quite short. The problem of WCAO analysis in operating systems is studied in [Lv09].

### *10.2.2 WCET of Complex Tasks*

We now turn to the WCET analysis of a preemptive complex task (C-task) that accesses protected shared objects. The WCET of such a task depends not only on behavior of the task itself, but also on the behavior of other tasks and the operating system of the node. WCET analysis of a C-task is therefore not a local problem of a single task, but a global problem involving all the interacting tasks within a node.

In addition to the delays caused by the task preemption (which was analyzed in the previous section), an additional delay that originates from the direct interactions caused by the intended task dependencies (mutual exclusion, precedence) must be considered. In the last few years, progress has been made in coping with the direct interactions caused by the intended task dependencies – e.g., access to protected shared objects controlled by the priority ceiling protocol [Sha94]. This topic will be investigated in Sect. 10.4.2 on the scheduling of dependent tasks.

### *10.2.3 Anytime Algorithms*

In practice, the time difference between the best-case execution time (BCET) and a guaranteed upper bound for the worst-case execution time (WCET) of a task can be substantial. *Anytime algorithms* are algorithms that use this time difference to improve the quality of the result as more execution time is provided [Chu08]. Anytime algorithms consist of a *root segment* that calculates a first approximation of the result of sufficient quality and a *periodic segment* that improves the quality of the previously calculated result. The periodic segment is executed repeatedly until the deadline, i.e. the guaranteed worst-case execution time of the root segment, is reached. Whenever the deadline occurs, the last version of the available result is delivered to the client. When scheduling an anytime algorithm, the completion of the root segment of the anytime algorithm must be guaranteed in order that a result of sufficient quality is available at this instant. The remaining time until the deadline is used to improve this result. The WCET problem of an anytime algorithm is thus reduced to finding a guaranteed upper bound for the WCET of the root segment. A loose upper bound of the WCET is of no serious concern, since the slack time between BCET and WCET is used to improve the result.

Most iterative algorithms are anytime algorithms. Anytime algorithms are used in pattern recognition, planning, and control.

An anytime algorithm should have the following properties [Zil96]:

1. Measurable quality: It must be possible to measure the quality of a result.
2. Monotonic: The quality of the result must be a non-decreasing function of time and should improve with every iteration.
3. Diminishing returns: The improvement of the result should get smaller as the number of iterations increases.
4. Interruptability: After the completion of the root segment, the algorithm can be interrupted at any time and deliver a reasonable result.

### *10.2.4 State of Practice*

The previous discussion shows that the analytic calculation of a reasonable upper WCET bound of an S-task which does not make use of operating system services is possible under restricting assumptions. There are a number of tools that support such an analysis [Wil08]. It requires an annotated source program that contains programmer-supplied application-specific information to ensure that the program terminates and a detailed model of the behavior of the hardware to achieve a reasonable upper WCET bound.

Bounds for the WCET of all time-critical tasks are *needed* in almost all hard real-time applications. This important problem is solved in practice by combining a number of diverse techniques:

1. Use of a restricted architecture that reduces the interactions among the tasks and facilitates the a priori analysis of the control structure. The number of explicit synchronization actions that require context switches and operating system services is minimized.
2. The design of WECT models and the analytic analysis of sub-problems (e.g., the maximum execution time analysis of the source program) such that an effective set of test cases biased towards the worst-case execution time can be generated automatically.
3. The controlled measurement of sub-systems (tasks, operating system service times) to gather experimental WCET data for the calibration of the WCET models.
4. The implementation of an *anytime algorithm*, where only a bound for WCET of the root segment must be provided.
5. The extensive testing of the complete implementation to validate the assumptions and to measure the safety margin between the assumed WCET and the actual measured execution times.

The state of current practice is not satisfactory, because in many cases the minimal and maximum execution times that are observed during testing are taken for the BCET and WCET. Such an observed upper bound cannot be considered a guaranteed upper bound. It is to be hoped that in the future the WCET problem will get easier, provided simple processors with private scratchpad memory will form the components of multi-processor systems-on-chips (MPSoCs).

## 10.3 Static Scheduling

In static or pre-runtime scheduling, a feasible schedule of a set of tasks is calculated off-line. The schedule must guarantee all deadlines, considering the resource, precedence, and synchronization requirements of all tasks. The construction of such a schedule can be considered as a constructive sufficient schedulability test. The precedence relations between the tasks executing in the different nodes can be depicted in the form of a *precedence graph* (Fig. 10.5).

### *10.3.1 Static Scheduling Viewed as a Search*

Static scheduling is based on strong regularity assumptions about the points in time when future service requests will be honored. Although the occurrence of external events that demand service is not under the control of the computer system, the recurring points in time when these events will be serviced can be established a priori by selecting an appropriate sampling rate for each class of events. During system design, it must be ascertained that the sum of the maximum delay times until a request is recognized by the system plus the maximum transaction response time is smaller than the specified service deadline.

*The Role of Time*. A static schedule is a periodic time-triggered schedule. The timeline is partitioned into a sequence of basic granules, the *basic cycle time*. There is only one interrupt in the system: the periodic clock interrupt denoting the start of a new basic granule. In a distributed system, this clock interrupt must be globally

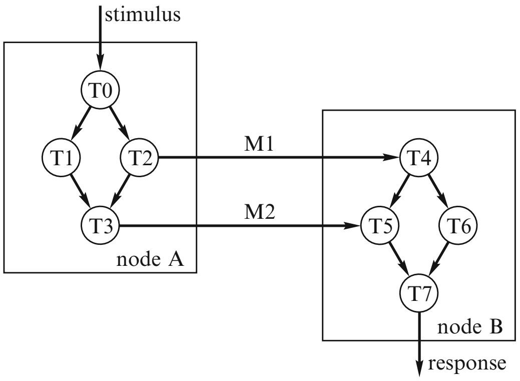


**Fig. 10.5** Example of a precedence graph of a distributed task set [Foh94]

synchronized to a precision that is much better than the duration of a basic granule. Every transaction is periodic, its period being a multiple of the basic granule. The least common multiple of all transaction periods is the *schedule period*. At compile time, the scheduling decision for every point of the schedule period must be determined and stored in a dispatcher table for the operating system for the full schedule period. At run time, the preplanned decision is executed by the dispatcher after every clock interrupt.

**Example:** If the periods of all tasks are harmonic, e.g., either a positive or negative power of two of the full second, the schedule period is equal to the period of the task with the longest period.

Static scheduling can be applied to a single processor, to a multiple-processor, or to a distributed system. In addition to preplanning the resource usage in all nodes, the access to the communication medium must also be preplanned in distributed systems. It is known that finding an optimal schedule in a distributed system is in almost all realistic scenarios an NP-complete problem, i.e., computationally intractable. But even a non-optimal solution is sufficient if it meets all deadlines.

*The Search Tree.* The solution to the scheduling problem can be seen as finding a path, a feasible schedule, in a *search tree* by applying a search strategy. An example of a simple search tree for the precedence graph of Fig. 10.5 is shown in Fig. 10.6. Every level of the search tree corresponds to one unit of time. The depth of the search tree corresponds to the period of the schedule. The search starts with an empty schedule at the root node of this tree. The outward edges of a node point to the possible alternatives that exist at this point of the search. A path from the root node to a particular node at level *n* records the sequence of scheduling decisions that have been made up to time-point *n*. Each path to a leaf node describes a complete schedule. It is the goal of the search to find a complete schedule that observes all precedence and mutual exclusion constraints, and which completes before the deadline. From Fig. 10.6, it can be seen that the right branch of the search tree will lead to a shorter overall execution time than the left branches.

*A Heuristic Function Guiding the Search.* To improve the efficiency of the search, it is necessary to guide the search by some heuristic function. Such a heuristic function can be composed of two terms, the actual cost of the path encountered until the present node in the search tree, i.e., the present point in the schedule, and the estimated cost until a goal node. Fohler [Foh94] proposes a heuristic function that estimates the time needed to complete the precedence graph, called TUR (time until

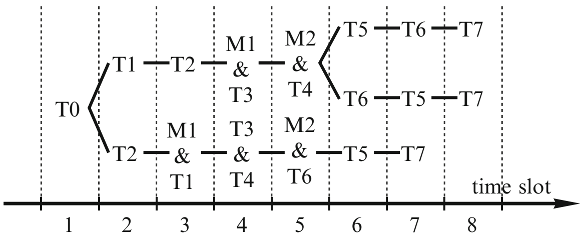


**Fig. 10.6** A search tree for the precedence graph of Fig. 10.5

response). A lower bound of the TUR can be derived by summing up the maximum execution times of all tasks and message exchanges between the current task and the last task in the precedence graph, assuming true parallelism constrained by the competition for CPU resources of tasks that reside at the same node. If this necessary TUR is not short enough to complete the precedence graph on time, all the branches from the current node can be pruned and the search must backtrack.

## 10.3.2 Increasing the Flexibility in Static Schedules

One of the weaknesses of static scheduling is the assumption of strictly periodic tasks. Although the majority of tasks in hard real-time applications is periodic, there are also sporadic service requests that have hard deadline requirements. An example of such a request is an *emergency stop* of a machine. Hopefully it will never be requested – the mean time between emergency stops can be very long. However, if an emergency stop is requested, it must be serviced within a small specified time interval.

The following three methods increase the flexibility of static scheduling:

1. The transformation of sporadic requests into periodic requests
2. The introduction of a sporadic server task
3. The execution of mode changes

*Transformation of a Sporadic Request to a Periodic Request.* While the future request times of a periodic task are known a priori, only the minimum inter-arrival time of a sporadic task is known in advance. The actual points in time when a sporadic task must be serviced are not known ahead of the request event. This limited information makes it difficult to schedule a sporadic request before run time. The most demanding sporadic requests are those that have a short response time, i.e., the corresponding service task has a low latency.

It is possible to find solutions to the scheduling problem if an independent sporadic task has a laxity $l$. One such solution, proposed by Mok [Mok93, p. 44], is the replacement of a sporadic task $T$ by a pseudo-periodic task $T'$ as seen in Table 10.1.

This transformation guarantees that the sporadic task will always meet its deadline if the pseudo-periodic task can be scheduled. The pseudo-periodic task can be scheduled statically. A sporadic task with a short latency will continuously demand a substantial fraction of the processing resources to guarantee its deadline, although it might request service very infrequently.

*Sporadic Server Task.* To reduce the large resource requirements of a pseudo-periodic task with a long inter-arrival time (period) but a short latency,

**Table 10.1** Parameters of the pseudo-periodic task

| Parameter | Sporadic task | Pseudo-periodic task |
|---|---|---|
| Computation time, $c$ | $c$ | $c' = c$ |
| Deadline interval, $d$ | $d$ | $d' = c$ |
| Period, $p$ | $p$ | $p' = \mathrm{Min}(l - 1, p)$ |

Sprunt et al. [Spr89] have proposed the introduction of a periodic server task for the service of sporadic requests. Whenever a sporadic request arrives during the period of the server task, it will be serviced with the high priority of the server task. The service of a sporadic request exhausts the execution time of the server. The execution time will be replenished after the period of the server. Thus, the server task preserves its execution time until it is needed by a sporadic request. The sporadic server task is scheduled dynamically in response to the sporadic request event.

*Mode Changes.* During the operation of most real-time applications, a number of different operating modes can be distinguished. Consider the example of a flight control system in an airplane. When a plane is taxiing on the ground, a different set of services is required than when the plane is flying. Better resource utilization can be realized if only those tasks that are needed in a particular operating mode must be scheduled. If the system leaves one operating mode and enters another, a corresponding change of schedules must take place.

During system design, one must identify all possible operating and emergency modes. For each mode, a static schedule that will meet all deadlines is calculated off-line. Mode changes are analyzed and the appropriate mode change schedules are developed. Whenever a mode change is requested at run time, the applicable mode change schedule will be activated immediately. We conclude this section with a comment by Xu and Parnas [Xu91, p. 134]:

> For satisfying timing constraints in hard real-time systems, predictability of the systems behavior is the most important concern; pre-run-time scheduling is often the only practical means of providing predictability in a complex system.

## 10.4 Dynamic Scheduling

After the occurrence of a significant event, a dynamic scheduling algorithm determines *on-line* which task out of the ready task set must be serviced next. The algorithms differ in the assumptions about the complexity of the task model and the future task behavior [But04].

### *10.4.1 Scheduling Independent Tasks*

The classic algorithm for scheduling a set of periodic independent hard real-time tasks in a system with a single CPU, the *rate monotonic algorithm*, was published in 1973 by [Liu73].

*Rate Monotonic Algorithm.* The rate monotonic algorithm is a dynamic preemptive algorithm based on static task priorities. It makes the following assumptions about the task set:

1. The requests for all tasks of the task set $\{T_i\}$ for which hard deadlines must be satisfied, are periodic.

2. All tasks are independent of each other. There exist no precedence constraints or mutual exclusion constraints between any pair of tasks.
3. The deadline interval of every task $T_i$ is equal to its period $p_i$.
4. The required maximum computation time of each task $c_i$ is known a priori and is constant.
5. The time required for context switching can be ignored.
6. The sum of the utilization factors $\mu$ of the $n$ tasks with period p is given by

$$\mu = \sum \mathrm{c_i}/\mathrm{p_i} \leqslant \mathrm{n}(2^{1/\mathrm{n}} - 1).$$

The term $\mathrm{n}(2^{1/n} - 1)$ approaches ln 2, i.e., about 0.7, as $n$ goes to infinity.

The rate monotonic algorithm assigns static priorities based on the task periods. The task with the shortest period gets the highest static priority, and the task with the longest period gets the lowest static priority. At run time, the dispatcher selects the task request with the highest static priority.

If all the assumptions are satisfied, the rate monotonic algorithm guarantees that all tasks will meet their deadline. The algorithm is optimal for single processor systems. The proof of this algorithm is based on the analysis of the behavior of the task set at the *critical instant*. A critical instant of a task is the moment at which the request of this task will have the longest response time. For the task system as a whole, the critical instant occurs when requests for all tasks are made simultaneously. Starting with the highest priority task, it can be shown that all tasks will meet their deadlines, even in the case of the critical instant. In a second phase of the proof it must be shown that any scenario can be handled if the *critical-instant scenario* can be handled. For the details of the proof refer to [Liu73].

It is also shown that assumption (6) above can be relaxed in case the task periods are harmonic, i.e., they are multiples of the period of the highest priority task. In this case the utilization factor $\mu$ of the $n$ tasks,

$$\mu = \sum \mathrm{c_i}/\mathrm{p_i} \leqslant 1,$$

can approach the theoretical maximum of unity in a single processor system.

In recent years, the rate monotonic theory has been extended to handle a set of tasks where the deadline interval can be different from the period [But04].

*Earliest-Deadline-First (EDF) Algorithm.* This algorithm is an optimal dynamic preemptive algorithm in single processor systems which are based on dynamic priorities. The assumptions (1) to (5) of the rate monotonic algorithm must hold. The processor utilization $\mu$ can go up to 1, even when the task periods are not multiples of the smallest period. After any significant event, the task with the earliest deadline is assigned the highest dynamic priority. The dispatcher operates in the same way as the dispatcher for the rate monotonic algorithm.

*Least-Laxity (LL) Algorithm.* In single processor systems, the least laxity algorithm is another optimal algorithm. It makes the same assumptions as the EDF algorithm. At any scheduling decision instant the task with the shortest laxity $l$, i.e., the difference between the deadline interval $d$ and the computation time $c$

$$d - c = l$$

is assigned the highest dynamic priority.

In multiprocessor systems, neither the earliest-deadline-first nor the least-laxity algorithm is optimal, although the least-laxity algorithm can handle task scenarios, which the earliest-deadline-first algorithm cannot handle and vice-versa.

### 10.4.2 Scheduling Dependent Tasks

From a practical point of view, results on how to schedule tasks with precedence and mutual exclusion constraints are much more important than the analysis of the independent task model. Normally, the concurrently executing tasks must exchange information and access common data resources to cooperate in the achievement of the overall system objectives. The observation of given precedence and mutual exclusion constraints is thus rather the norm than the exception in distributed real-time systems.

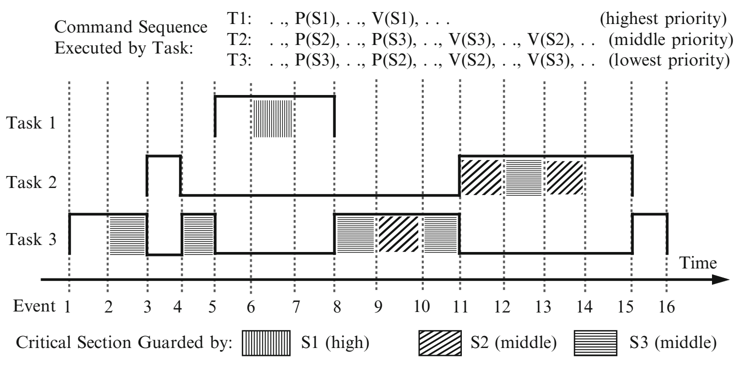
To solve this problem, the *priority ceiling protocol* was developed by [Sha90]. The *priority ceiling protocol* can be used to schedule a set of periodic tasks that have exclusive access to common resources protected by semaphores. These common resources, e.g., common data structures, can be utilized to realize an inter-task communication.

The priority ceiling of a semaphore is defined as the priority of the highest priority task that may lock this semaphore. A task T is allowed to enter a critical section only if its assigned priority is higher than the priority ceilings of all semaphores currently locked by tasks other than T. Task T runs at its assigned priority unless it is in a critical section and blocks higher priority tasks. In this case it inherits the highest priority of the tasks it blocks. When it exits, the critical section it resumes the priority it had at the point of entry into the critical section.

The example of Fig. 10.7, taken from [Sha90], illustrates the operation of the priority ceiling protocol. A system of 3 tasks: $T1$ (highest priority), $T2$ (middle priority), and $T3$ (lowest priority) compete for three critical regions protected by the three semaphores $S1$, $S2$ and $S3$.

*Schedulability Test for the Priority Ceiling Protocol.* The following sufficient schedulability test for the priority ceiling protocol has been given by [Sha90]. Assume a set of periodic tasks, $\{T_i\}$ with periods $p_i$ and computation times $c_i$. We denote the worst-case blocking time of a task $t_i$ by lower priority tasks by $B_i$. The set of $n$ periodic tasks $\{T_i\}$ can be scheduled, if the following set of inequalities holds, $\forall$ $i$, $1 \leq i \leq n$.

$$(\mathrm{c}_1/\mathrm{p}_1 + \mathrm{c}_2/\mathrm{p}_2 + \cdots + \mathrm{c}_\mathrm{i}/\mathrm{p}_\mathrm{i} + \mathrm{B}_\mathrm{i}/\mathrm{p}_\mathrm{i}) \leqslant \mathrm{i}(2^{1/\mathrm{i}} - 1)$$


Command Sequence Executed by Task:
T1: . ., P(S1), . ., V(S1), . . . (highest priority)
T2: . ., P(S2), . ., P(S3), . ., V(S3), . ., V(S2), . . (middle priority)
T3: . ., P(S3), . ., P(S2), . ., V(S2), . ., V(S3), . . (lowest priority)
Task 1
Task 2
Task 3
Time
Event 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Critical Section Guarded by:
S1 (high)
S2 (middle)
S3 (middle)

| Event | Action |
|---|---|
| 1 | T3 begins execution. |
| 2 | T3 locks S3. |
| 3 | T2 is started and preempts T3. |
| 4 | T2 becomes blocked when trying to access S2 since the priority of T2 is not higher than the priority ceiling of the locked S3. T3 resumes the execution of its critical section at the inherited priority of T2. |
| 5 | T1 is initiated and preempts T3. |
| 6 | T1 locks the semaphore S1. The priority of T1 is higher than the priority ceiling of all locked semaphores. |
| 7 | T1 unlocks semaphore S1. |
| 8 | T1 finishes its execution. T3 continues with the inherited priority of T2. |
| 9 | T3 locks semaphore S2. |
| 10 | T3 unlocks S2. |
| 11 | T3 unlocks S3 and returns to its lowest priority. At this point T2 can lock S2. |
| 12 | T2 locks S3. |
| 13 | T2 unlocks S3. |
| 14 | T2 unlocks S2. |
| 15 | T2 completes. T3 resumes its operation. |
| 16 | T3 completes. |

**Fig. 10.7** The priority ceiling protocol (example taken from [Sha90])

In these inequalities the effect of preemptions by higher priority tasks is considered in the first *i* terms (in analogy to the rate monotonic algorithm), whereas the worst case blocking time due to all lower priority tasks is represented in the term $B_i/p_i$. The blocking term $B_i/p_i$, which can become very significant if a task with a short period (i.e., small $p_i$) is blocked for a significant fraction of its time, effectively reduces the CPU utilization of the task system. In case this first sufficient schedulability test fails, more complex sufficient tests can be found in [Sha90]. The priority ceiling protocol is a good example of a predictable, but *non-deterministic* scheduling protocol.

**Example:** The NASA Pathfinder robot on MARS experienced a problem that was diagnosed as a classic case of priority inversion, due to a missing *priority ceiling protocol*. The full and most interesting story is contained in [Jon97]:

> Very infrequently it was possible for an interrupt to occur that caused the (medium priority) communications task to be scheduled during the short interval while the (high priority) information bus thread was blocked waiting for the (low priority) meteorological data thread. In this case, the long-running communications task, having higher priority than the meteorological task, would prevent it from running, consequently preventing the blocked information bus task from running. After some time had passed, a watchdog timer would go off, notice that the data bus task had not been executed for some time, conclude that something had gone drastically wrong, and initiate a total system reset.

## 10.5 Alternative Scheduling Strategies

### *10.5.1 Scheduling in Distributed Systems*

In a control system, the maximum duration of an *RT transaction* is the critical parameter for the quality of control, since it contributes to the dead time of a control loop. In a distributed system, the duration of this transaction depends on the sum of the durations of all processing and communication actions that form the transaction. In such a system, it makes sense to develop a *holistic schedule* that considers all these actions together. In a time-triggered system, the processing actions and the communication actions can be *phase aligned* (see Sect. 3.3.4), such that a send slot in the communication system is available immediately after the WCET of a processing action.

It is already difficult to guarantee tight deadlines by dynamic scheduling techniques in a single processor event-triggered multi-tasking system if mutual exclusion and precedence constraints among the tasks must be considered. The situation is more complex in a distributed system, where the non-preemptive access to the communication medium must be considered. Tindell [Tin95] analyzes distributed systems that use the CAN bus as the communication channel and establishes analytical upper bounds to the communication delays that are encountered by a set of periodic messages. These results are then integrated with the results of the node-local task scheduling to arrive at the worst-case execution time of distributed real-time transactions. One difficult problem is the control of transaction jitter.

Since the worst-case duration of a RT transaction in an event-triggered distributed system can be exceedingly pessimistic, some researchers are looking at dynamic best-effort strategies and try to establish bounds based on a probabilistic analysis of the scheduling problem. This approach is not recommended in hard-real time systems, since the characterization of *rare events* is extremely difficult. Rare event occurrences in the environment, e.g., a lightning stroke into an electric power grid, will cause a highly correlated input load on the system (e.g., an alarm shower) that is very difficult to model adequately. Even an extended observation of a real-life system is not conclusive, because these rare events, by definition, cannot be observed frequently.

In soft real-time systems (such as multimedia systems) where the occasional miss of deadline is tolerable, probabilistic analysis is widely used. An excellent survey on the results of 25 years of research on real-time scheduling is contained in [Sha04].

### *10.5.2 Feedback Scheduling*

The concept of feedback, well established in many fields of engineering, uses information about the *actual behavior* of a scheduling system to dynamically adapt the *scheduling algorithms* such that the intended behavior is achieved. Feedback scheduling starts with the establishment and observation of relevant performance parameters of the scheduling system. In a multimedia systems, the queue size that develops before a server process is an example of such a relevant performance parameter. These queue sizes are continuously monitored and the producer of information is controlled – either slowed down or speeded up – in order to keep the size of the queue between given levels, the *low* and *high watermark*.

By looking at the scheduling problem and control problem in an integrated fashion, better overall results can be achieved in many control scenarios. For example, the sample rate of a process can be dynamically adjusted based on the observed performance of the physical process.

## Points to Remember

- A scheduler is called *dynamic* (or *on-line*) if it makes its scheduling decisions at run time, selecting one out of the current set of *ready* tasks. A scheduler is called static (or pre-run-time) if it makes its scheduling decisions at compile time. It generates a dispatching table for the run-time dispatcher off-line.
- A test that determines whether a set of ready tasks can be scheduled so that each task meets its deadline is called a *schedulability test*. We distinguish between *exact*, *necessary*, and *sufficient* schedulability tests. In nearly all cases of task dependency, even if there is only one common resource, the complexity of an exact schedulability test algorithm belongs to the class of NP-complete problems and is thus computationally intractable.
- While the future request times of a periodic task are known a priori, only the minimum interarrival time of a *sporadic* task is known in advance. The actual points in time when a sporadic task must be serviced are not known ahead of the request event.
- The *adversary argument* states that, in general, it is not possible to construct an optimal totally on-line dynamic scheduler if there are mutual exclusion constraints between a periodic and a sporadic task. The adversary argument accentuates the value of a priori information about the behavior in the future.
- In general, the problem of determining the worst-case execution time (WCET) of an arbitrary sequential program is unsolvable and is equivalent to the halting problem for Turing machines. The WCET problem can only be solved, if the programmer provides additional application-specific information at the source code level.

- In static or pre-run-time scheduling, a feasible schedule of a set of tasks that guarantees all deadlines, considering the resource, precedence, and synchronization requirements of all tasks, is calculated off-line. The construction of such a schedule can be considered as a constructive sufficient schedulability test.
- The rate monotonic algorithm is a dynamic preemptive scheduling algorithm based on static task priorities. It assumes a set of periodic and independent tasks with deadlines equal to their periods.
- The Earliest-Deadline-First (EDF) algorithm is a dynamic preemptive scheduling algorithm based on dynamic task priorities. The task with the earliest deadline is assigned the highest dynamic priority.
- The Least-Laxity (LL) algorithm is a dynamic preemptive scheduling algorithm based on dynamic task priorities. The task with the shortest laxity is assigned the highest dynamic priority.
- The priority ceiling protocol is used to schedule a set of periodic tasks that have exclusive access to common resources protected by semaphores. The *priority ceiling* of a semaphore is defined as the priority of the highest priority task that may lock this semaphore.
- According to the priority ceiling protocol, a task T is allowed to enter a critical section only if its assigned priority is higher than the priority ceilings of all semaphores currently locked by tasks other than T. Task T runs at its assigned priority unless it is in a critical section and blocks higher priority tasks. In this case, it inherits the highest priority of the tasks it blocks. When it exits the critical section, it resumes the priority it had at the point of entry into the critical section.
- The critical issue in best-effort scheduling concerns the assumptions about the input distribution. Rare event occurrences in the environment will cause a highly correlated input load on the system that is difficult to model adequately. Even an extended observation of a real-life system is not conclusive, because these rare events, by definition, cannot be observed frequently.
- In soft real-time systems (such as multimedia systems) where the occasional miss of deadline is tolerable, probabilistic scheduling strategies are widely used.
- Anytime algorithms are algorithms that improve the quality of the result as more execution time is provided. They consist of a *root segment* that calculates a first approximation of the result of sufficient quality and a *periodic segment* that improves the quality of the previously calculated result.
- In *feedback scheduling*, information about the *actual behavior* of a scheduling system is used to dynamically adapt the *scheduling algorithms* such that the intended behavior is achieved.

## Bibliographic Notes

Starting with the seminal work of Liu and Layland [Liu73] in 1973 on scheduling of independent tasks, hundreds of papers on scheduling are being published each year. In 2004 the *Real-Time System Journal* published a comprehensive survey

*Real-Time Scheduling Theory: A Historical Perspective* [Sha04]. The book by Butazzo [But04] *Hard Real-Time Computer Systems* covers scheduling extensively. An excellent survey article on WCET analysis is contained in [Wil08]. Anytime algorithms are described in [Zil96].

## Review Questions and Problems

10.1 Give taxonomy of scheduling algorithms.

10.2 Develop some necessary schedulability tests for scheduling a set of tasks on a single processor system.

10.3 What are the differences between periodic tasks, sporadic tasks, and aperiodic tasks?

10.4 Why is it hard to find the worst-case execution time (WCET) of a program?

10.5 What is the worst-case administrative overhead (WCAO)?

10.6 Given the following set of independent periodic tasks, where the deadline interval is equal to the period: {T1(5,8); T2(2,9); T3(4,13)}; (notation: task name(CPU time, period)).

(a) Calculate the laxities of these tasks.
(b) Determine, using a necessary schedulability test, if this task set is schedulable on a single processor system.
(c) Schedule this task set on a two-processor system with the LL algorithm.

10.7 Given the following set of independent periodic tasks, where the deadline interval is equal to the period: {T1(5,8); T2(1,9); T3(1,5)}; (notation: task name(CPU time, period)).

(a) Why is this task set not schedulable with the rate monotonic algorithm on a single processor system?
(b) Schedule this task set on a single processor system with the EDF algorithm.

10.8 Why is it not possible to design, in general, an optimal dynamic scheduler?

10.9 Assume that the task set of Fig. 10.5 is executed without the priority ceiling protocol. At what moment will a deadlock occur? Can this deadlock be resolved by priority inheritance? Determine the point where the priority ceiling protocol prevents a task from entering a critical section.

10.10 Discuss the schedulability test of the priority ceiling protocol. What is the effect of blocking on the processor utilization?

10.11 What are the problems with dynamic scheduling in distributed systems?

10.12 Discuss the issue of temporal performance in best-effort distributed system.

10.13 What is the role of time in static scheduling?

10.14 List some properties of anytime algorithms!

10.15 What is feedback scheduling?