2

TERMINOLOGY AND ASSUMPTIONS

Real-time scheduling involves the allocation of resources and time intervals to tasks in such a way that certain timeliness performance requirements are met. Scheduling has been perhaps the most widely researched topic within real-time systems. This is due to the belief that the basic problem in real-time systems is to make sure that tasks meet their time constraints. This Chapter introduces basic terminology, assumptions, notation, and metrics necessary to fully understand the remaining Chapters of the book.

It should be mentioned that two different research communities have examined real-time scheduling problems from their own perspectives. Scheduling in the Operations Research (OR) community has focussed on job-shop and flow-shop problems, with and without deadlines. For instance, manpower scheduling, project scheduling, and scheduling of machines are some of the topics studied in OR [5, 6, 7, 8]. The types of resources assumed by OR researchers (machines, factory cells, etc.) and how jobs use those resources (e.g., a job may be required to use every machine in some specified order) are quite different from those assumed by Computer Science researchers (CPU cycles, memory, etc. and where jobs typically use only a single machine). Activities on a factory floor typically have larger time granularities than those studied by computer scientists. The metrics of interest to the OR community such as: minimizing maximum cost, minimizing the sum of completion times, minimizing schedule length, minimizing tardiness, and minimizing the number of tardy jobs are often not of interest to real-time system designers. Rather, real-time system designers attempt to prove all tasks meet their deadlines, or in less stringent situations, they try to minimize the number of tasks which miss their deadlines. OR techniques are geared towards static (off-line) methods where those developed in Computer

Science focus more on dynamic techniques. In this book scheduling problems are examined from the perspective of Computer Science.

2.1 TASK MODELS, ASSUMPTIONS AND NOTATION

Real-time systems can be quite complex with many different types of tasks, time and reliability requirements, and metrics. The basic terminology, assumptions and notation used throughout the book are now defined and described. Other terminology is introduced in later Chapters when it applies to a particular algorithm or system configuration.

Definition 2.1 *A* real-time task *is an executable entity of work which, at a minimum, is characterized by a worst case execution time and a time constraint.*

Definition 2.2 *A* job *is an instance of a task.*

There are three types of real-time tasks: periodic, aperiodic, and sporadic. Each type normally gives rise to multiple jobs.

Definition 2.3 Periodic tasks *are real-time tasks which are activated (released) regularly at fixed rates (period). In keeping with common notation, the period is designated by T. Normally, periodic tasks have constraints which indicate that instances of them must execute once per period T. The time constraint for a periodic task is a deadline d that can be less than, equal to, or greater than the period. The most common case is when the deadline equals the period.*

Definition 2.4 Synchronous periodic tasks *are a set of periodic tasks where all first instances are released at the same time, usually considered time zero.*

Definition 2.5 Asynchronous periodic tasks *are a set of periodic tasks where tasks can have their first instance released at different times.*

Definition 2.6 Aperiodic tasks *are real-time tasks which are activated irregularly at some unknown and possibly unbounded rate. The time constraint is usually a deadline d.*

Definition 2.7 Sporadic tasks *are real-time tasks which are activated irregularly with some known bounded rate. The bounded rate is characterized by a* minimum interarrival period, *that is, a minimum interval of time between two successive activations. This is necessary (and achieved by some form of flow control) in order to bound the workload generated by such tasks. The time constraint is usually a deadline d.*

Definition 2.8 *A* hybrid task set *is a task set containing both periodic and sporadic tasks.*

Time constraints can be release times or deadlines, or both.

Definition 2.9 *A* release time, *r, is a point in time at which a real-time job becomes ready to (or is activated to) execute.*

Definition 2.10 *A* deadline, *d is a point in time by which the task (job) must complete.*

Usually, a deadline d is an absolute time. Sometimes, d is also used to refer to a relative deadline when there is no confusion. To emphasize relative deadlines D is used. The deadline can be hard, soft, or firm.

Definition 2.11 *A* hard deadline *means that it is vital for the safety of the system that this deadline is always met.*

Definition 2.12 *A* soft deadline *means that it is desirable to finish executing the task (job) by the deadline, but no catastrophe occurs if there is a late completion.*

Definition 2.13 *A* firm deadline *means that a task should complete by the deadline, or not execute at all. There is no value to completing the task after its deadline.*

Accordingly, real-time tasks are often distinguished as *hard*, *soft*, and *firm* tasks. Sometimes, soft tasks do not have deadlines at all. Their requirement is then to complete as soon as possible.

Scheduling constraints are sometimes expressed with respect to tasks, when they do not depend on particular instances, with respect to jobs, otherwise.

With these definitions presented, the notation used in the remainder of the book is as follows. The i^{th} task in the system is denoted by τ_i. Its j^{th} instance is denoted by $J_{i,j}$. Sometimes it is only important to distinguish jobs and it is not important what tasks they are an instance of. In these cases, J_i is used to denote the $i(th)$ unique job. In two clearly marked areas of the book, J is redefined to mean the amount of jitter that a job experiences. This should not cause any confusion. Each task usually has a worst case or maximum execution time C_i. A periodic task has also a period denoted by T_i. The minimum interarrival time of sporadic tasks is also designated by T_i.

Definition 2.14 *A job has* release time r *if its execution can begin only at time* $t \geq r$.

Definition 2.15 *A job has* deadline d *if its execution must complete by* d.

The release time of the j^{th} job of the periodic task τ_i is most commonly given as:

$$r_{i,j} = (j-1)T_i,$$

and its deadline is:

$$d_{i,j} = r_{i,j} + T_i = jT_i,$$

that is, the deadline of one instance is the release time of the next instance.

For sporadic tasks, the assumption is that the release times of two consecutive instances must be separated by at least its minimum interarrival time, that is:

$$r_{i,j} \geq r_{i,j-1} + T_i.$$

The deadline is often assumed to be equal to the earliest possible release time of the next instance, that is:

$$d_{i,j} = r_{i,j} + T_i.$$

An example of an EDF schedule is depicted in Figure 2.1. The first task in the schedule, which is sporadic, has maximum execution time 2 and minimum interarrival time 5. The example shows 2 arrivals of T_1 at times 5 and 17.

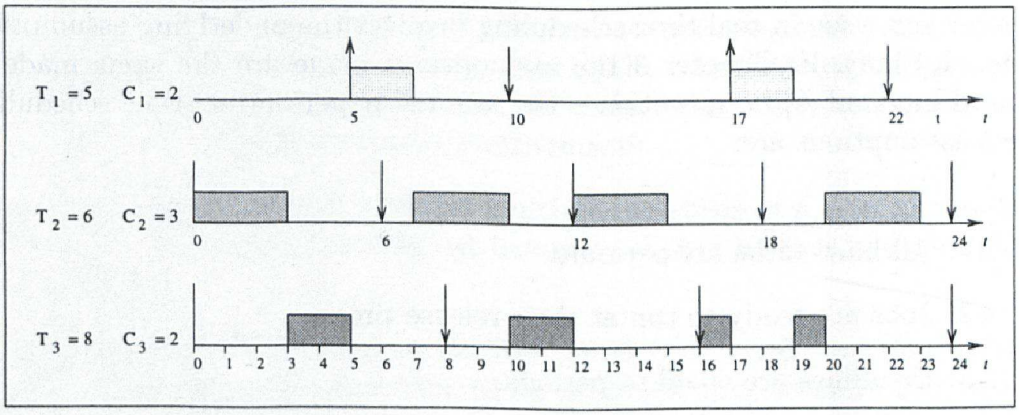

Figure 2.1 Example of EDF schedule.

The other two tasks are periodic and they have maximum execution times 3 and 2, and periods 6 and 8, respectively. The schedule is represented on three horizontal time axes, one for each task. Along the axes, release times of sporadic task instances are represented by upward arrows, while deadlines are represented by downward arrows. For instance, the first job of τ_1 has release time 5 and deadline 10.

Deadlines of periodic task instances are represented by downward arrows, as well. Release times are usually represented by a vertical segment. When they coincide with deadlines and there is no ambiguity they are not shown. For instance, the first job of τ_2 is released at time $t = 0$ and has deadline 6. 6 is also the release time of the second job.

The assignment of jobs to the processor is represented by filled rectangular boxes drawn along the axes. For instance, at time $t = 0$ the job with the earliest deadline in the system is $J_{2,1}$. This job gets the processor and completes at time $t = 3$. At this point the processor is assigned to $J_{3,1}$.

An example of preemption is represented at time $t = 17$. At time $t = 16$ the processor is assigned to $J_{3,3}$. At time $t = 17$ $J_{1,2}$ is released. $J_{1,2}$ has deadline 22 and becomes the job with the earliest deadline in the system (i.e., 22), hence it preempts the execution of $J_{3,3}$ and gets the processor. When it completes at time $t = 19$, $J_{3,3}$ can resume its execution.

Note that meanwhile, $J_{2,4}$ has also been released. However, its deadline is equal to the deadline of $J_{3,3}$ and it is executed later. Actually, ties could be broken arbitrarily.

Another key issue in real-time scheduling involves the underlying assumptions made. Initially, in Chapter 3 the assumptions made are the same made by Liu and Layland [9] because this is the seminal paper on real-time scheduling. These assumptions are:

- A1: All hard tasks are periodic.
- A2: Jobs are ready to run at their release times.
- A3: Deadlines are equal to periods.
- A4: Jobs do not suspend themselves.
- A5: Jobs are independent in that there are neither synchronizations between them, nor shared resources other than the CPU, nor relative dependencies or constraints on release times or completion times.
- A6: There are no overhead costs for preemption, scheduling, or interrupt handling.
- A7: Processing is fully preemptable at any point.

These assumptions are acceptable for a first step in the study of real-time scheduling theory. However, they are not practical and not adequate for the analysis of most actual systems. For this reason, one goal of this book is to present results in which one or more of these assumptions is relaxed. For example, in addition to timing constraints, a task may also possess the following types of constraints and requirements:

- Resource constraints – A task may require access to certain resources other than the CPU, such as, I/O devices, networking, data structures, files, and databases [4, 11].
- Precedence relationships – A complex task, for example, one requiring access to many resources, is better handled by breaking it up into multiple subtasks related by precedence constraints and each requiring a subset of the resources.
- Concurrency constraints – Tasks are allowed concurrent access to resources providing the consistency of the resources is not violated.

- Communication requirements – Sets of cooperating tasks are the norm for distributed, hard real–time systems. The communication requirements are a function of the semantics of the communication (synchronous, asynchronous) and of their timing requirements.
- Placement constraints – When multiple instances of a task are executed for fault-tolerance, the different instances should be executed on different processors.
- Criticalness – Depending on the functionality of a task, meeting the deadline of one task may be considered more critical than another. For example, a task that reacts to an emergency situation, such as, a fire on the factory floor is more critical than the task that controls the movements of a robot under normal operating conditions.

2.2 STATIC VERSUS DYNAMIC SCHEDULING

Most classical scheduling theory deals with static scheduling. Static scheduling refers to the fact that the scheduling algorithm has complete knowledge regarding the task set and its constraints, such as, deadlines, computation times, precedence constraints, and future release times. This set of assumptions is realistic for many real-time systems. For example, real-time control of a simple laboratory experiment or a simple process control application might have a fixed set of sensors and actuators, and a well defined environment and set of processing requirements. In these types of real-time systems, the static scheduling algorithm operates on this set of tasks and produces a single schedule that is fixed for all time. Sometimes there is confusion regarding future release times. If all future release times are known when the algorithm is developing the schedule then it is still a static algorithm.

In contrast, a dynamic scheduling algorithm (in the context of this book) has complete knowledge of the currently active set of tasks, but new arrivals may occur in the future, not known to the algorithm at the time it is scheduling the current set. The schedule therefore changes over time. Dynamic scheduling is required for real-time systems such as teams of robots cleaning up a chemical spill or in military command and control applications. Fewer theoretical results are known about real-time dynamic scheduling algorithms than for static algorithms.

Off-line scheduling is often equated to static scheduling, but this is wrong. In building any real-time system, off-line scheduling (analysis) should always be done regardless of whether the final runtime algorithm is static or dynamic. In many real-time systems, the designers can identify the maximum set of tasks with their worst case assumptions and apply a static scheduling algorithm to produce a static schedule. This schedule is then fixed and used on-line with well understood properties such as, given that all the assumptions remain true, all tasks meet the deadlines. In other cases, the off-line analysis might produce a static set of priorities to use at runtime. The schedule itself is not fixed, but the priorities that drive the schedule are fixed. This is common in the rate monotonic approach.

If the real-time system is operating in a more dynamic environment, then it is not feasible to meet the assumptions of static scheduling (i.e., everything is known *a priori*). In this case an algorithm is chosen and analyzed off-line for the expected dynamic environmental conditions. Usually, less precise statements about the overall performance can be made. On-line, this same dynamic algorithm executes.

Generally, a scheduling algorithm (possibly with some modifications) can be applied to static scheduling or dynamic scheduling and used off-line or on-line. The important difference is what is known about the performance of the algorithm in each of these cases. As an example, consider earliest deadline first (EDF) scheduling. When applied to static scheduling it is known that EDF is optimal in many situations (to be enumerated in this book), but when applied to dynamic scheduling on multi-processors it is not optimal, in fact, it is known that no algorithm can be optimal.

Predictability is one of the primary issues in real-time systems. Schedulability analysis or feasibility checking of the tasks of a real-time system has to be done to predict whether the tasks meet their timing constraints. Several scheduling paradigms emerge, depending on (a) whether a system performs schedulability analysis, (b) if it does, whether it is done statically or dynamically, and (c) whether the result of the analysis itself produces a schedule or plan according to which tasks are dispatched at run-time. Based on this the following classes of algorithms are identified:

- *Static table-driven approaches:* These perform static schedulability analysis and the resulting schedule (or table, as it is usually called) is used at run-time to decide *when* a task must begin execution.

- *Static priority-driven preemptive approaches*: These perform static schedulability analysis, but unlike in the previous approach, no explicit schedule is constructed. At run-time, tasks are executed highest-priority-first.
- *Dynamic planning-based approaches*: Unlike the previous two approaches, feasibility is checked at run-time, i.e., a dynamically arriving task is accepted for execution only if it is found feasible, i.e., will make its deadline. Such a task is said to be *guaranteed* to meet its time constraints. This is sometimes called *admission control*. One of the results of the feasibility analysis is a schedule or plan that is used to decide when a task can begin execution. However, similar to the static case, the feasibility check and schedule creation can be separated. For example, in classical real-time systems it has been common that a schedule is created, while in real-time multimedia scheduling it is common to separate the feasibility check from the scheduling.
- *Dynamic best-effort approaches*: In this approach no feasibility checking is done. The system tries to do its best to meet deadlines. But since no guarantees are provided, a task may be aborted during its execution.

It must be pointed out that even though four categories have been identified, some scheduling techniques possess characteristics that span multiple paradigms. Each of these categories is now briefly elaborated.

Static table-driven approaches are applicable to tasks that are periodic (or have been transformed into periodic tasks by well known techniques). Given task characteristics, a table is constructed, using one of many possible techniques (e.g., using various search heuristics), that identifies the start and completion times of each task and tasks are dispatched according to this table. This is a highly predictable approach but, is highly inflexible since any change to the tasks and their characteristics may require a complete overhaul of the table.

The approach traditionally used in non real-time systems is the *priority-based preemptive scheduling* approach. Here, tasks have priorities that may be statically or dynamically assigned and at any time, the task with the highest priority executes. It is the latter requirement that necessitates preemption: if a low priority task is in execution and a higher priority task arrives, the former is preempted and the processor is given to the new arrival. If priorities are assigned systematically in such a way that timing constraints can be taken into account, then the resulting scheduler can also be used for real-time systems. For example, using the rate-monotonic approach [9], utilization bounds can be derived such that if a set of tasks do not exceed the bound, they can

be scheduled without missing any deadlines using such a *static priority-driven preemptive scheduler.*

Cyclic scheduling, used in many large-scale dynamic real-time systems [3], is a combination of both table-driven scheduling and priority scheduling. Here, tasks are assigned one of a set of harmonic periods. Within each period, tasks are dispatched according to a table that just lists the order in which the tasks execute. It is slightly more flexible than the table-driven approach because no start times are specified and it is amenable to *a priori* bound analysis – if maximum requirements of tasks in each cycle are known beforehand. However, pessimistic assumptions are necessary for determining these requirements. In many actual applications, rather than making worse-case assumptions, confidence in a cyclic schedule is obtained by very elaborate and extensive simulations of typical scenarios. This approach is both error-prone and expensive [10].

The *dynamic planning-based approaches* provide the flexibility of dynamic approaches with some of the predictability of approaches that check for feasibility. Here, after a task arrives, but before its execution begins, an attempt is made to create a schedule that contains the previously guaranteed tasks as well as the new arrival. If the attempt fails and if the attempt is made sufficiently ahead of the deadline, time is available to take alternative actions. This approach provides for predictability with respect to individual arrivals.

In contrast, if a purely priority-driven preemptive approach is used, say, by using task deadlines as priorities, and without any planning, a task could be preempted any time during its execution. In this case, until the deadline arrives, or until the task finishes, whichever comes first, it is not known whether the timing constraint is met. This is the major disadvantage of the *dynamic best-effort approaches.* If, however, the worst case performance characteristics of such a scheduler can be analyzed, then perhaps it can be recognized and avoided. Such worst case analyses are in their infancy, being applicable only to tasks with very simple characteristics [2].

2.3 METRICS

Classical scheduling theory typically uses metrics such as minimizing the sum of completion times, minimizing the weighted sum of completion times, minimizing schedule length, minimizing the number of processors required, or mini-

mizing the maximum lateness. In most cases, deadlines are not even considered in these results. When deadlines are considered, they are usually added as constraints, where, for example, one creates a minimum schedule length, subject to the constraint that all tasks must meet their respective deadlines.

If one or more tasks miss their deadlines, then there is no feasible solution. Which of these classical metrics (where deadlines are not included as constraints) are of most interest to real-time systems designers? The sum of completion times is generally not of interest because there is no direct assessment of timing properties (deadlines or periods). However, the weighted sum is very important when tasks have different values that they impart to the system upon completion. Using value is often (erroneously) overlooked in many real-time systems where the focus is simply on deadlines and not on a combination of value and deadline. Minimizing schedule length has secondary importance in possibly helping minimize the resources required for a system, but does not directly address the fact that individual tasks have deadlines. The same is true for minimizing the number of processors required. Minimizing the maximum lateness metric can be useful at design time where resources can be continually added until the maximum lateness is equal to zero. In this case no tasks miss their deadlines. On the other hand, the metric is not always useful because minimizing the maximum lateness doesn't necessarily prevent one, many, or even ALL tasks from missing their deadlines.

In the static real-time scheduling problem, an off-line schedule is to be found that meets all deadlines. If more than one such schedule exists, a secondary metric, such as maximizing the average *earliness* is used to choose one among them. When a task completes its earliness is the amount of time still remaining before its deadline. If no such schedule exists, one which minimizes the average *tardiness or lateness* may be chosen. Tardiness is the amount of time by which a task misses its deadline. In these cases, an algorithm's ability to achieve optimality is with respect to these secondary metrics.

In real-time systems, scheduling results are often presented in terms of *schedulability* or *feasibility* analysis.

Definition 2.16 *A set of jobs is* schedulable *or* feasible *if all timing constraints are met, that is, all hard real-time jobs complete by their respective deadlines.*

Definition 2.17 An optimal real-time scheduling algorithm *is one which may fail to meet a deadline only if no other scheduling algorithm can meet the deadline.*

This definition of optimality is the typical one used in real-time scheduling. A common (non real-time) definition of optimality says that an algorithm is optimal if it minimizes (maximizes) some cost function. It is important to be familiar with both definitions.

In dynamic real-time systems, since, in general, it cannot be *a priori* guaranteed that all deadlines are met, maximizing the number of arrivals that meet their deadlines is often used as a metric. Some of the results presented utilize the metric of minimizing the number of tasks that miss their deadlines which is the dual of maximizing the number that meet their deadlines.

The variety of metrics that have been suggested for real-time systems is indicative of the different types of real-time systems that exist in the real world as well as the types of requirements imposed on them. This sometimes makes it hard to compare different scheduling algorithms.

Related to metrics is the complexity of the various scheduling problems themselves. Many scheduling problems are NP-complete or NP-hard [4]. NP is the class of all decision problems that can be solved in polynomial time by a nondeterministic machine. A recognition problem R is NP-complete if $R \in NP$ and all other problems in NP are polynomial transformable to R. A recognition or optimization problem R is NP-hard if all problems in NP are polynomial transformable to R, but it can't be shown that $R \in NP$. The complexity of the various problems presented in this book is mentioned throughout. The reader should take special note throughout the text regarding the types of tasks' constraints that move the scheduling problem from P to NP, e.g., in some problem situations allowing preemption moves a problem from NP-hard to polynomial and in other problems adding a release time constraint might move the problem from polynomial to NP-hard.

REFERENCES

[1] N. Audsley, A. Burns, M. Richardson, and A. Wellings, "Hard Real-Time Scheduling: The Deadline Monotonic Approach," *IEEE Workshop on Real-Time Operating Systems*, 1992.

[2] S. Baruah, G. Koren, D. Mao, B. Mishra, A. Raghunathan, L. Rosier, D. Shasha, and F. Wang, "On the Competitiveness of On-Line Real-Time Task Scheduling," *Proceedings of Real-Time Systems Symposium*, December 1991.

[3] G. Carlow, "Architecture of the Space Shuttle Primary Avionics Software System," *CACM*, 27(9), September 1984.

[4] R. Garey and D. Johnson, "Complexity Results for Multiprocessor Scheduling Under Resource Constraints," *SIAM Journal of Computing*, 1975.

[5] R. Graham, Bounds on the Performance of Scheduling Algorithms, chapter in *Computer and Job Shop Scheduling Theory*, John Wiley and Sons, pp. 165-227, 1976.

[6] E. Lawler, "Optimal Sequencing of a Single Machine Subject to Precedence Constraints," *Management Science*, 19, 1973.

[7] E.L. Lawler, "Recent Results in the Theory of Machine Scheduling," Mathematical Programming: the State of the Art, A. Bachen et al. (eds.), Springer-Verlag, New York, 1983.

[8] J. Lenstra and A. Rinnooy Kan "Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey," *Ann. Discrete Math.* 5, pp. 287-326, 1977.

[9] C. Liu and J. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *Journal of the Association for Computing Machinery* **20**(1), 1973.

[10] D. Locke, "Software Architectures for Hard Real-Time Applications: Cyclic Executives versus Fixed Priority Executives," *Real-Time Systems*, Vol. 4, No. 2, March 1992.

[11] W. Zhao, K. Ramamritham and J. Stankovic, "Scheduling Tasks with Resource Requirements in Hard Real-Time Systems, " *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 5, 1987.