Chapter 2
Preliminaries: Workload and Platform Models

Multiprocessor real-time scheduling theory studies the scheduling of real-time workloads upon multiprocessor platforms. This chapter describes some of the models that are currently widely used for representing such workloads and platforms, and provides some explanation and justification for the use of these particular models.

2.1 Workload Models

In choosing a model to represent a hard-real-time computer application system, the application system's designers are faced with two—often contradictory—concerns. On the one hand, they would like the model to be *general*, in order that it may accurately reflect the relevant characteristics of the application system being modeled. On the other hand, it is necessary that the model be *efficiently analyzable*[1], if it is to be helpful in system design and analysis.

Over the years, attempts to balance these two different goals have resulted in various models being proposed for representing real-time workloads. While these models differ considerably from each other in their expressive power and in the computational complexity of their associated analysis problems, most of them share some common characteristics.

The workload is modeled as being comprised of basic units of work known as *jobs*. Each job is characterized by three parameters: an *arrival time* or *release date*; a *worst-case execution time (WCET)*; and a *deadline*, with the interpretation that it may need to execute for an amount up to its WCET within a *scheduling window* that spans the time interval within its release date and its deadline (see Fig. 2.1). Each job is assumed to represent a single thread of computation; hence, a job may execute upon at most one processor at any instant.

Much of the real-time scheduling theory deals with systems in which the jobs are generated by a finite collection of independent recurrent tasks.

[1] Recall our discussion in Sect. 1.5: we seek models for which the analysis problems of interest can be solved in polynomial or pseudo-polynomial time.

S. Baruah et al., *Multiprocessor Scheduling for Real-Time Systems,*
Embedded Systems, DOI 10.1007/978-3-319-08696-5_2

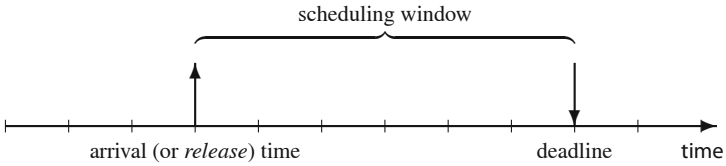

Fig. 2.1 Jobs: terminology. A job is characterized by an arrival (or release) time, a worst-case execution time, and a deadline. The time-interval between its release time and its deadline is called its scheduling window. It needs to execute for an amount up to its worst-case execution time within its scheduling window

The different tasks are *independent*[2] in the sense that the parameters of the jobs generated by each task are completely independent of the other tasks in the system. Different models for hard-real-time tasks place different constraints upon the permissible values for the parameters of the jobs generated by each task. We focus here on what are known as *sporadic* task models. In such models, lower bounds are specified between arrivals of successive jobs of the same task; by contrast, *periodic* and some other models specify exact separations, or upper bounds on these separations. In this book, we will look at different sporadic task models. The *three-parameter model* is the most widely-studied one; it is described in Sect. 2.1.2. A *directed acyclic graph (DAG)-based model* has recently come in for some attention in the context of multiprocessor systems, because it is better able to model parallelism within workloads that may be exploited upon multiprocessor platforms; we introduce this model in Sect. 2.1.3, but a more detailed discussion is postponed to Chap. 21. We start out in Chaps. 5–9 with an exploration of a restricted version of the three-parameter model that is commonly referred to as the *Liu and Layland (LL)* model. Much of the rest of this book is focused upon the 3-parameter model; we provide a summary of results concerning the DAG-based model in Chap. 21.

2.1.1 The Liu and Layland (LL) Task Model

This task model was formally defined in a paper [139] coauthored by C. L. Liu and J. Layland; hence the name. In this model, a task is characterized by two parameters—a *worst-case execution requirement* (WCET) C_i and a *period* (or *inter-arrival separation parameter*) T_i. A Liu and Layland task denoted τ_i is thus represented by an ordered pair of parameters: $\tau_i = (C_i, T_i)$. Such a task generates a potentially infinite sequence of jobs. The first job may arrive at any instant, and the arrival times of any two successive jobs are at least T_i time units apart. Each job has a WCET of C_i, and a deadline that is T_i time units after its arrival time. A Liu and Layland *task system* consists of a finite number of such Liu and Layland tasks executing upon a shared platform.

[2] This independence assumption represents a tradeoff between expressiveness and tractability of analysis; this trade-off is discussed in Sect. 2.3.

2.1.2 The Three-Parameter Sporadic Tasks Model

This model was proposed as a generalization to the Liu and Layland task model. As indicated by the name of the model, each task in this model [149] is characterized by three parameters: a *relative deadline* D_i in addition to the two parameters—WCET C_i and period T_i—that characterize Liu and Layland tasks. A 3-parameter sporadic task denoted by τ_i is thus represented by a 3-tuple of parameters: $\tau_i = (C_i, D_i, T_i)$. Such a task generates a potentially infinite sequence of jobs. The first job may arrive at any instant, and the arrival times of any two successive jobs are at least T_i time units apart. Each job has a WCET of C_i, and a deadline that occurs D_i time units after its arrival time. A 3-parameter sporadic *task system* consists of a finite number of such 3-parameter sporadic tasks executing upon a shared platform. A task system is often denoted as τ, and described by enumerating the tasks in it: $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$.

By allowing for the specification of a relative deadline parameter in addition to a period, the 3-parameter sporadic tasks model offers a means of specifying recurrent workloads that may occur infrequently (large period), but are urgent (have a small deadline). This is the model for recurrent tasks that has been most widely studied by the real-time scheduling theory community; for the most part in this book when we talk of "sporadic tasks" we mean tasks represented in this model.

Depending upon the relationship between the values of the relative deadline and period parameters of the tasks in it, a 3-parameter sporadic task system may further be classified as follows:

- In an *implicit-deadline* task system, the relative deadline of each task is equal to the task's period: $D_i = T_i$ for all $\tau_i \in \tau$. (Implicit-deadline task systems are exactly the same as Liu and Layland task systems).
- In a *constrained-deadline* task system, the relative deadline of each task is no larger than the task's period: $D_i \leq T_i$ for all $\tau_i \in \tau$.
- Tasks in an *arbitrary-deadline* task system do not need to have their relative deadlines satisfy any constraint with regards to their periods.

It is evident from these definitions that each implicit-deadline task system is also a constrained-deadline task system, and each constrained-deadline task system is also an arbitrary-deadline task system.

The ratio of the WCET of the task to its period is called the *utilization* of the task, and the ratio of the WCET to the smaller of its period and relative deadline is called its *density*. The *utilization of a task system* is defined to be the sum of the utilizations of all the tasks in the system. We will use the following notation in this book to represent the utilizations and densities of individual sporadic tasks, and of sporadic task systems.

Utilization: The utilization u_i of a task τ_i is the ratio C_i / T_i of its WCET to its period. The total utilization $U_{\text{sum}}(\tau)$ and the largest utilization $U_{\text{max}}(\tau)$ of a task system τ are defined as follows:

$$U_{\text{sum}}(\tau) \stackrel{\text{def}}{=} \sum_{\tau_i \in \tau} u_i; \qquad U_{\text{max}}(\tau) \stackrel{\text{def}}{=} \max_{\tau_i \in \tau} (u_i)$$

Density: The density dens_i of a task τ_i is the ratio $(C_i/\min(D_i, T_i))$ of its WCET to the smaller of its relative deadline and its period. The total density $dens_{\text{sum}}(\tau)$ and the largest density $dens_{\max}(\tau)$ of a task system τ are defined as follows:

$$\text{dens}_{\text{sum}}(\tau) \stackrel{\text{def}}{=} \sum_{\tau_i \in \tau} \text{dens}_i; \qquad \text{dens}_{\max}(\tau) \stackrel{\text{def}}{=} \max_{\tau_i \in \tau} (\text{dens}_i)$$

Parallel Execution The different tasks in a task system are assumed, for the most part, to be independent of each other; hence, jobs of different tasks may execute simultaneously on different processors upon a multiprocessor platform. Parallelism within a single job is forbidden in our model: each job is assumed to represent a single thread of computation, which may execute upon at most one processor at any point in time.

As the scheduling windows of different jobs of the same task do not overlap in implicit-deadline and constrained-deadline task systems, each task in such a system executes upon at most one processor at each instant in time.

For tasks in arbitrary-deadline task systems that have relative deadline greater than period, the scheduling windows of multiple successive jobs of the task may overlap. The default assumption for such tasks is that multiple jobs of the same task may *not* execute simultaneously: a job must complete execution before the next job of the task begins to execute.

Thus, the 3-parameter sporadic task model does not in general allow for the modeling of parallelism within a single task. This has traditionally not been perceived as a shortcoming of the model, since the model was first developed in the context of uniprocessor systems, for which parallel execution is not possible in any case.

More recently, however, the trend toward implementing real-time systems upon multiprocessor and multicore platforms has given rise to a need for models that are capable of exposing any possible parallelism that may exist within the workload to the scheduling mechanism. Therefore, there has been a move in the real-time scheduling community toward considering new models that allow for partial parallelism within a task, as well as for precedence dependencies between different parts of each individual task. We describe such a model in Sect. 2.1.3 below.

2.1.3 The Sporadic DAG Tasks Model

Each recurrent task in this model is modeled as a DAG $G_i = (V_i, E_i)$, a (relative) deadline parameter D_i, and a period T_i. Each vertex $v \in V_i$ of the DAG corresponds to a sequential job of the kind discussed above, and is characterized by a WCET. Each (directed) edge of the DAG represents a precedence constraint: if (v, w) is a (directed) edge in the DAG then the job corresponding to vertex v must complete execution before the job corresponding to vertex w may begin execution. Groups of jobs that are not constrained (directly or indirectly) by precedence constraints in such a manner may execute in parallel if there are processors available for them to do so.

The task is said to release a *dag-job* at time-instant t when it becomes available for execution. When this happens, all $|V_i|$ of the jobs are assumed to become available for execution simultaneously, subject to the precedence constraints. The task may release an unbounded sequence of dag-jobs during runtime; all $|V_i|$ jobs that are released at some time-instant t must complete execution by time-instant $t + D_i$. A minimum interval of duration T_i must elapse between successive releases of dag-jobs. If $D_i > T_i$ it is not required that all jobs of a particular dag-job complete execution before execution of jobs of the next dag-job may begin—i.e., no precedence constraints are assumed between the jobs of different dag-jobs.

The sporadic DAGs task model will be discussed further in Chap. 21.

2.2 A Taxonomy of Multiprocessor Platforms

Various details must be provided in order to completely specify a multiprocessor platform. These include—how many processors comprise the platform? Are all these processors of the same kind, or do they possess different computing capabilities? How are the processors connected to each other? etc. In addition, we must specify whether the platform supports *preemption* and inter-processor *migration*.

As stated in Sect. 1.2, there is a classification of multiprocessor platforms as *identical*, *uniform*, or *unrelated*, depending upon the relative computing capabilities of the different processors; we will, for the most part, focus upon multiprocessor platforms comprised of multiple identical processors (we will briefly summarize some results concerning real-time scheduling upon unrelated multiprocessors in Chap. 22). We address the remaining questions below.

2.2.1 Preemption and Migration

Preemptive scheduling permits that a job executing upon a processor may be interrupted by the scheduler (perhaps because the scheduler needs to execute some other job), and have its execution resumed at a later point in time.

In nonpreemptive scheduling, such preemption is forbidden: once a job begins execution, it continues to execute until it has completed. (In *limited preemptive* scheduling, various kinds of restrictions are placed upon the occurrence of preemptions during scheduling).

Results concerning preemptive scheduling are typically obtained under the simplifying assumption that a preemption incurs no cost. As stated in Sect. 1.5, we will primarily study preemptive scheduling in this book.

Different algorithms for scheduling systems of recurrent real-time tasks upon multi-processor platforms place different restrictions as to where different tasks' jobs are permitted to execute. A *global* scheduling algorithm allows any job to execute upon any processor; by contrast, a *partitioned* scheduling algorithm first maps each recurrent task on to a particular processor, and only executes jobs of task upon

the processor to which it has been mapped. Global and partitioned scheduling may both be considered as special cases of *clustered* scheduling, in which the processors comprising the multiprocessor platform are partitioned into clusters, and each recurrent task mapped on to a single cluster. Migration of a task's jobs is only allowed within the cluster to which the task is mapped.

In addition to the family of scheduling algorithms that may be considered to be specialized forms of clustered scheduling, there are scheduling algorithms that place various forms of restriction upon migration without forbidding it outright. Such algorithms are commonly called *semi-partitioned* or *limited migrative* scheduling algorithms; they may, for example, specify that no individual task is allowed to migrate between more than two processors, or they may restrict the total number of migratory tasks in a system, etc.

Inspired by some features that are available in modern multiprocessor operating systems such as Linux, some work has recently been done [39, 104] on a migrative model that is called the *processor affinities* model. In this model, a set of processors upon which a task may execute is specified for each task, and jobs of the task are allowed to migrate amongst these processors. The processor affinities model differs from clustered scheduling in that the sets of processors specified for the different tasks may overlap and hence not constitute a partitioning or a clustering of the set of available processors.

Although some interesting results concerning limited-migrative scheduling and the use of processor affinities have been obtained recently, we do not cover these paradigms; for the most part, the focus of this book remains on partitioned and global scheduling. We briefly state a few results concerning the semi-partitioned scheduling of Liu & Layland task systems in Sect. 6.5, and mention an interesting extension of partitioned scheduling called *federated scheduling* in the context of scheduling systems of sporadic DAG tasks, in Sect. 21.2.

2.3 Discussion

In this section, we attempt to justify some of the restrictions we have chosen to place upon the range of workload and machine models to be covered in this book. We particularly address the following questions:

1. Why do we primarily restrict ourselves to *sporadic* task models?
2. Why do we require *independence* amongst the tasks?
3. What are the consequences of ignoring preemption and migration costs?

2.3.1 The Restriction to Sporadic Task Models

We had stated, in Sect. 1.3, that we would be considering the scheduling of sporadic, rather than periodic, task systems in this book. As stated there, the main distinguishing feature of sporadic task systems is that *minimum*, rather than exact, inter-arrival

separations are specified between the arrival of different pieces of work (jobs) from a task.

From a pragmatic perspective, our decision to restrict attention to sporadic models is driven by our desire, also stated in Chap. 1 (in Sect. 1.5), to only consider problems for which solutions are tractable—have polynomial or pseudo-polynomial run-time. It appears that defining a recurrent task model that violates the sporadic assumption in any meaningful way results in scheduling analysis problems that are highly intractable (nondeterministic polynomial time (NP)-hard in the strong sense)—an illustrative example is provided in Sect. 4.4, where it is shown that a very basic analysis problem for periodic task systems is highly intractable even upon single-processor platforms. Thus, our desire for efficiently solvable analysis problems restricts our choice of task models, and it appears that the various sporadic task models are about as general as one can get without running up against the intractability barrier (the separation between tractable and intractable models for uniprocessor scheduling is very precisely and methodically demarcated in Stigge's dissertation [168]; see also [169]).

From an application perspective, too, an argument can be made in favor of considering sporadic task models. Note that in sporadic task models, the different tasks do not really need a common notion of global time; the only requirement is that they all share a common notion of duration (i.e., they should all agree on the duration of "real time" represented by a unit of time). By contrast, stricter notions of recurrence such as periodic tasks assume that the different tasks generate jobs at specific "global" *instants* in time; one consequence of this is that periodic and similar models are more sensitive to differences between the notions of time maintained by different sources that may be responsible for generating the jobs of different tasks. Consider, for example, a distributed system in which each task (i.e., the associated process) maintains its own (very accurate) clock, and in which the clocks of different tasks are not synchronized with each other. The accuracy of the clocks permit us to assume that there is no clock drift, and that all tasks use exactly the same units for measuring time. However, the fact that these clocks are not synchronized rules out the use of a concept of an absolute time scale. This idea is explored further in Sect. 2.3.2 below, as the second task independence assumption.

We note that, since periodic behavior is one of the possible behaviors of a sporadic task system with corresponding parameter values, the worst-case behavior of a sporadic task system "covers" the behavior of the corresponding periodic task system; therefore, showing that a sporadic task system will always meet all deadlines implies that the corresponding periodic task system will also do so (although the converse of this statement is not true: a periodic task system may meet all deadlines but the corresponding sporadic task system may not).

A further note: upon uniprocessor platforms, the difference between periodic and sporadic behavior is sometimes brushed aside (see, e.g., [140, p. 40], which claims that scheduling analysis results for periodic task models remain valid even if the period parameters are reinterpreted to represent minimum inter-arrival separations). The rationale for this (backed by some empirical evidence) is that the worst-case behavior of the sporadic task system is evidenced when it behaves as a periodic system;

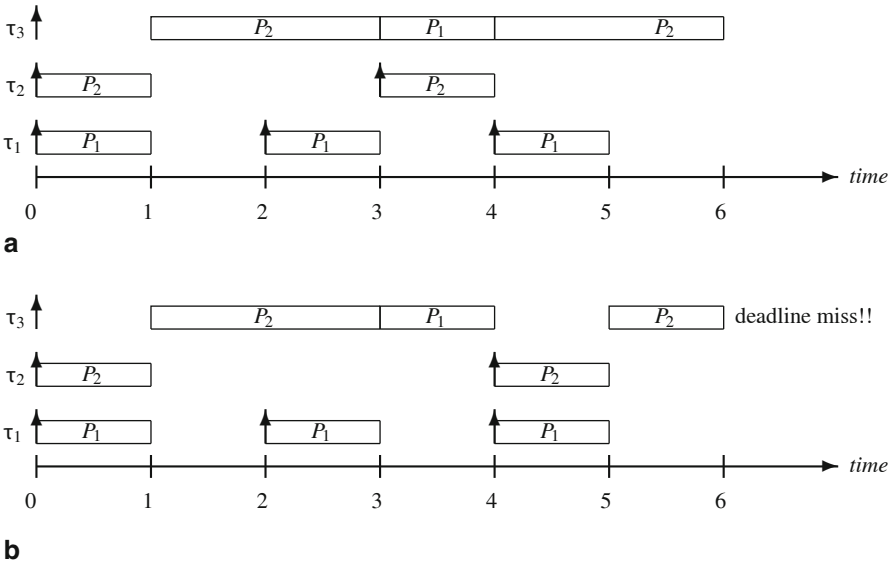

Fig. 2.2 **a** A 2-processor schedule for the job arrivals of task system τ of Example 2.1, interpreted as a *periodic* task system—repeat this schedule over all intervals $[6k, 6(k+1))$ for all $k = 0, 1, 2, \ldots$. Each up arrow denotes a job arrival: job-arrivals of τ_1 (at time-instants 0, 2, and 5) are depicted by the bottom-most row of up arrows, job-arrivals of τ_2 (at time-instants 0 and 3) by the middle row of up arrows, and the sole job-arrival of τ_3 (at time-instant 0) is depicted by the up arrow on the topmost row. **b** An infeasible sequence of job arrivals of the same task system interpreted as a sporadic task system

for example, it is known that a system of 3-parameter sporadic tasks is *infeasible*—cannot always be scheduled to meet all deadlines—on a single processor if and only if the arrival sequence for the equivalent periodic task system is also infeasible. But this is not necessarily true upon multiprocessors; consider the following example.

Example 2.1 As shown in Fig. 2.2, the task system τ comprised of the following three tasks

$$\tau_1 = (1, 1, 2), \tau_2 = (1, 1, 3), \tau_3 = (5, 6, 6)$$

is feasible on two processors under global scheduling if interpreted as periodic tasks, but not if interpreted as sporadic tasks (we cannot meet all deadlines for the sporadic task system if τ_1's second job arrives 3 time units after the first rather than arriving the minimum of 2 time units after).

2.3.2 The Task Independence Assumption

A pair of conditions collectively called the *task independence assumptions* were specified in [41]. These assumptions, listed below, dictate the process by which jobs

are generated by the tasks in the system; once generated, the jobs (each characterized by an arrival time, an execution requirement, and a deadline) are independent of each other. That is, while the task independence assumptions restrict the job-generation process, they make no assertions about the interactions of the jobs once they have been generated.

1. *The runtime behavior of a task should not depend upon the behavior of other tasks.* That is, each task is an independent entity, perhaps driven by separate external events. It is not permissible for one task to generate a job directly in response to another task generating a job.
2. *The workload constraints are specified without referencing "absolute" time.* Hence, specifications such as "Task T generates a job at time-instant 3" are forbidden.
   (Note that periodic task systems in which an initial *offset* is specified for each task—see Sect. 4.4—violate the task independence assumption since these offsets are defined in terms of an absolute time scale).

In terms of sets of jobs that may legally be generated by a task system, the first task independence assumption implies that a set of jobs generated by an entire task system is legal in the context of the task system if and only if the jobs generated by each task are legal with respect to the constraint associated with that task. Examples of task systems *not* satisfying this assumption include systems where, for example, all tasks are required to generate jobs at the same time instant, or where it is guaranteed that certain tasks will generate jobs before certain other tasks. (To represent such systems in a manner satisfying this assumption, the interacting tasks must instead be modeled as a single task which is assumed to generate the jobs actually generated by the interacting tasks).

Letting an ordered 3-tuple (a, e, d) represent a job with arrival time a, execution requirement e, and deadline d, the second task independence assumption implies that if $\{(a_o, e_o, d_o), (a_1, e_1, d_1), (a_2, e_2, d_2) \dots\}$ is a legal arrival set with respect to the workload constraint for some task in the system, then so is the set $\{(a_o - x, e_o, d_o - x), (a_1 - x, e_1, d_1 - x), (a_2 - x, e_2, d_2 - x) \dots\}$, where x may be any real number.

The task independence assumptions are extremely general and are satisfied by a wide variety of the kinds of task systems one may encounter in practice. The various flavors of sporadic task systems discussed in Sect. 2.1 above satisfy these assumptions, as do "worst-case" periodic task systems—periodic task systems where each task may choose its offset.

So does a distributed system in which each task executes on a separate node (jobs correspond to requests for time on a shared resource), and which begins execution in response to an external event. All temporal specifications are made relative to the time at which the task begins execution, which is not *a priori* known.

It is noteworthy that answering interesting schedulability-analysis questions (such as determining feasibility) for many nontrivial task systems not satisfying the task independence assumptions (such as periodic task systems with deadlines not equal to period) turns out to be computationally difficult (often NP-hard in the strong sense), and hence of limited interest from the perspective of efficient analysis.

2.3.3 Preemption and Migration Costs

As stated in Sect. 2.2.1 above, in this book we are, for the most part, assuming a preemptive model of computation: a job executing on the processor may have its execution interrupted at any instant in time and resumed later, with no cost or penalty. In a similar vein, when we permit inter-processor migration in global scheduling, we assume that there is no cost or penalty associated with such migration. These are both clear approximations: in most actual systems, we would expect both a preemption and a migration to incur some delay and computational cost (we recommend the excellent discussions in Brandenburg's dissertation [65] for a detailed look at some of the issues arising in attempting to account for such overhead costs in actual computer systems).

There is a class of scheduling algorithms called *priority-driven* algorithms (Definition 3.3) that we will discuss in greater detail in Chap. 3; for now, it suffices to state that very many of the scheduling algorithms that we will look at in this book are priority-driven ones. Priority-driven algorithms possess the pleasing properties that

1. The number of preemptions in any preemptive schedule generated by such algorithms is strictly less than the number of jobs that are being scheduled, and
2. The number of job interprocessor migrations in any preemptive global schedule generated by such algorithms is also strictly less than the number of jobs that are being scheduled.

Hence, one can account for the overhead cost of preemptions and migrations by simply inflating the WCET parameters by the worst-case cost of one preemption and one migration—we will discuss this idea in further detail in Chap. 3.

Sources

A useful discussion of models (albeit primarily in the context of uniprocessor scheduling) is to be found in the survey paper [169] and in Stigge's dissertation [168]. The taxonomy of multiprocessor platforms may be found in textbooks on scheduling in the Operations Research context (where *processors* are commonly called *machines*). The task independence assumptions are from [41].

http://www.springer.com/978-3-319-08695-8

Multiprocessor Scheduling for Real-Time Systems

Baruah, S.; Bertogna, M.; Buttazzo, G.

2015, XV, 228 p. 45 illus., 4 illus. in color., Hardcover
ISBN: 978-3-319-08695-8