8

Real-Time Scheduling

We have so far studied a model-based approach to the design and analysis of embedded systems. In this chapter, we turn our attention to a key aspect of *implementing* embedded systems so that the implementation exhibits the intended timing behavior. As an example, consider the event-triggered component `MeasureSpeed` of figure 2.30. To implement this component, whenever one of the input events *Second* or *Rotate* is detected, the update code of its task needs to be executed. While defining the execution semantics of synchronous models, we assumed that a task executes instantaneously. Whether such an assumption is justified for a given implementation depends on the answers to a number of questions: How long does it take to execute the code of this task on the underlying processor? Does the task `MeasureSpeed` have its own dedicated processor, or are multiple tasks sharing the same processor? Is the task `MeasureSpeed` independent of the other tasks, or does it have to be executed only after some other task has finished executing? The theory of real-time scheduling focuses on the formalization of demands for processing time by different computational tasks and general policies for allocating processing time so that these demands are met. This subject has a rich history with applications to safety-critical embedded systems as well as signal processing and multimedia systems. In this chapter, we first introduce the most commonly occurring pattern for demand for processing time and then study two classical and widely used algorithms for real-time scheduling.

8.1 Scheduling Concepts

For the purpose of scheduling decisions concerning allocation of processing time, the basic unit of computation is called a *job*. Examples of jobs include execution of the task corresponding to the component `MeasureSpeed` of figure 2.30 and execution of the code corresponding to a mode-switch in the sender process in the timed audio control protocol of figure 7.13. In multimedia applications such as processing of the incoming video stream, a job can correspond to decoding

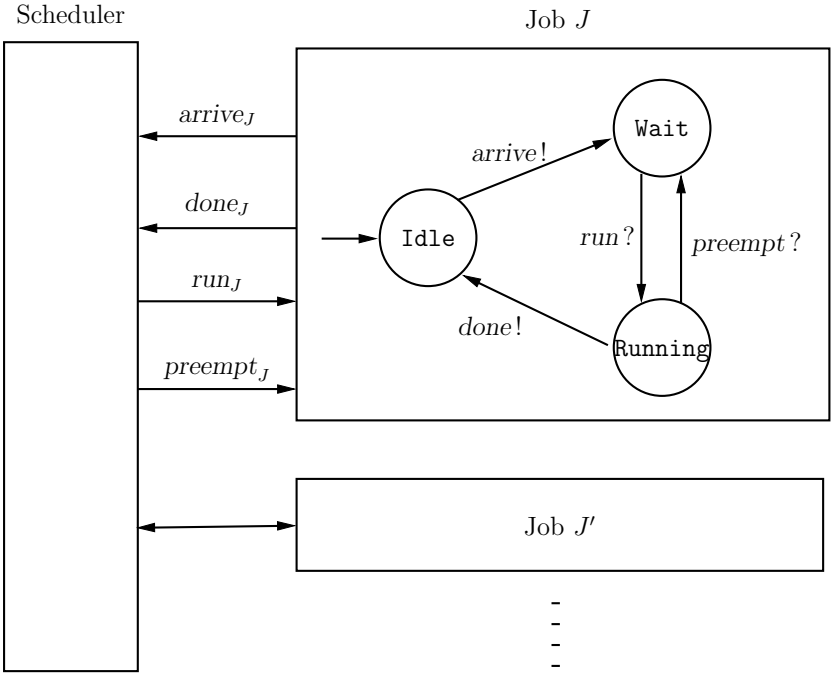

Figure 8.1: Interaction between the Scheduler and the Jobs

of a video frame in an MPEG file, while in a real-time control application such as avionics, a job can correspond to converting an analog signal from a sensor to a discrete value meaningful to the control software.

8.1.1 Scheduler Architecture

Figure 8.1 shows a typical interaction pattern between the scheduler and different jobs. Each job J is an independent process that communicates with the scheduler process responsible for allocation of processing time via events. The illustration also shows an abstract view of how the status of a job changes as a state machine.

Initially, a job J is in the mode `Idle`. When a job needs processing time, it communicates with the scheduler using the event $arrive_J$, and its status changes to `Wait`. When the scheduler decides to allocate the processor to the job J, it notifies the job using the event run_J, and this changes its mode to `Running`. When the current instance of the job J finishes its execution, it communicates with the scheduler using the event $done_J$ and returns to the mode `Idle`. The subsequent instance of the same job can now again request processing time using the event $arrive_J$.

When a job J is running, the scheduler may decide to *preempt* it before its computation is finished and allocate the processor to another job J'. The event $preempt_J$ switches the mode of the job J from `Running` to `Wait`, where it continues to wait for the scheduler to issue another run_J event.

The scheduler has two requirements. First, the processor can be allocated to only one job at any point in time, and thus at most one job should be in the mode `Running` at any time. Second, each instance of the job should get "enough" computation time. To formalize this requirement, we need to know the timing pattern of the arrivals of successive instances of each job, how much computation time each job instance needs, and by when each instance needs to finish its execution.

The model of timed processes studied in chapter 7 is rich enough to formalize the arrival pattern and usage requirements for the jobs, the interaction between a job and the scheduler, and the decision logic of the scheduler. However, in a typical implementation, the scheduler is an integral part of the operating system and has tight control over the execution of the jobs. Furthermore, the time needed to execute the decision logic used by the scheduler is much smaller compared with the demands for execution time by the jobs. As a result, we assume that processing time is divided into discrete time slots. All the interaction between the scheduler and all the jobs happens instantaneously at the beginning of each time slot. The demand for processing time by each job is specified in units of time slots (for example, by specifying the bounds on the number of time slots between arrivals of successive instances of a job), and the allocation strategy of the scheduler is completely specified by an assignment of time slots to the jobs. Such an allocation scheme is called *time-triggered* allocation and is an example of a computation model that is synchronous and timed. In this scheme, the choice of the length of a single time slot—whether it is, for example, a second or a millisecond, does not matter for designing resource-allocation policies as long as all the parameters of all the jobs are specified using this length as the basic time unit.

8.1.2 Periodic Job Model

A job model specifies the arrival pattern and usage requirements for the jobs. The most common such job model is the *periodic job model*, in which the demand for processing time is specified using three parameters: a period, a deadline, and a worst-case execution time.

Period

In the periodic model, each job J has an associated period $\pi(J)$, which is a positive number. This specifies that the job J is to be executed periodically every $\pi(J)$ time units, that is, the event $arrive_J$ is issued every $\pi(J)$ time units starting at time 0. Since a periodic job J is to be executed repeatedly, we use the notation J^a, for every positive integer a, to denote the ath instance of the

job. The ath instance of the job J is ready to be executed at time $(a-1)*\pi(J)$, and this time is called the *arrival time* of the ath instance, denoted $\alpha(J,a)$.

Deadline

Each job J has an associated deadline $\delta(J)$, which is a positive number, that specifies that each instance of the job must finish its execution within $\delta(J)$ time units of its arrival. In other words, the delay between the occurrence of an event $arrive_J$ and the subsequent event $done_J$ should be bounded by $\delta(J)$ time units. It is required that this relative deadline should not exceed the period: the condition $\delta(J) \leq \pi(J)$ should hold. Since the arrival time of the ath instance of the job J is $(a-1)*\pi(J)$, this instance should finish execution by the deadline $(a-1)*\pi(J)+\delta(J)$, and we denote this absolute deadline for the ath instance of the job J by $\delta(J,a)$. For example, if a job has period 5 and deadline 4, then the arrival time of its third instance is 10, and the deadline for this instance is 14.

When the deadline equals the period, it means that each instance of the job should finish executing before the arrival of the next instance. Such a deadline is called an *implicit deadline*. While implicit deadlines are common, allowing a deadline to be strictly smaller than the period allows specification of more stringent timing requirements, as meeting such explicit deadlines implies improved response time.

Worst-Case Execution Time

While the period specifies how often a job needs to be executed, and the deadline specifies by when each job instance needs to finish executing, the (worst-case) execution time specifies how long it takes to execute an instance of a job. Each job J has an associated worst-case execution time (WCET) $\eta(J)$, which is a positive number, such that the execution of an instance of a job takes at most $\eta(J)$ time units. In other words, the job J is guaranteed to issue the event $done_J$ if it spends a total cumulative time of $\eta(J)$ time units in the mode `Running` since the last occurrence of its arrival. Note that $\eta(J)$ is an *upper bound* on how long the computation corresponding to an instance of the job J can take to execute on the given platform. The actual execution time may vary, but if the scheduler allocates $\eta(J)$ time units to an instance before its deadline, then this allocation policy is safe. Since $\delta(J)$ specifies the deadline by which a job instance should finish executing, if the WCET $\eta(J)$ exceeds $\delta(J)$, then it is clear that the deadline cannot be met. Henceforth, we will assume that the condition $\eta(J) \leq \delta(J) \leq \pi(J)$ holds.

The three parameters $\eta(J)$, $\delta(J)$, and $\pi(J)$ together specify the requirement that, for every positive integer a, the job J should be allowed to execute for a total of $\eta(J)$ time units between the arrival time $\alpha(J,a) = (a-1)*\pi(J)$ of the instance J^a, and the deadline $\delta(J,a) = (a-1)*\pi(J)+\delta(J)$ for this instance. For example, if a job has period 5, deadline 4, and WCET 3, then it should

be allocated three time units between times 0 and 4, three time units between times 5 and 9, three time units between times 10 and 14, and so on.

Estimating WCET

The period and the deadline for a job follow from the design-time requirements of a system. The WCET, however, is an artifact of the software implementation and the execution platform, and we would like an analysis tool to derive this bound automatically by analyzing the code. Deriving WCET bounds is an active area of research with a variety of approaches. Let us review the basic idea underlying the approach based on *statically* analyzing the code, that is, by examining the syntactic structure of the code without actually having to execute it.

Let us assume that the code corresponding to a job is loop-free and consists of atomic assignment statements, conditional statements, and sequences of statements. This is indeed the assumption we have been using for the update code of tasks used in the earlier chapters. Suppose we know how to associate an execution time $\eta(stmt)$ with an atomic statement $stmt$ of the form $x := e$ and an execution time $\eta(e)$ for evaluating a Boolean expression e. Then the following two rules can be used to associate the WCET with a block of code:

1. **Sequencing:** If a statement $stmt$ is the sequence $(stmt_1; stmt_2; \ldots stmt_l)$ of l statements, then the execution time of $stmt$ is the sum of execution times of the component statements:

$$\eta(stmt) \;=\; \eta(stmt_1) + \eta(stmt_2) + \cdots + \eta(stmt_l).$$

2. **Conditional statement:** If a statement $stmt$ is the conditional statement (`if` e `then` $stmt_1$ `else` $stmt_2$), then the execution time of $stmt$ is the sum of the execution time of evaluating the test e and the maximum of the execution times of the statements $stmt_1$ and $stmt_2$:

$$\eta(stmt) \;=\; \eta(e) \;+\; \texttt{max}\,\{\,\eta(stmt_1), \eta(stmt_2)\,\}.$$

To understand the rule for conditional statements, observe that to execute the conditional statement, first the test e is evaluated, and then either of the two statements $stmt_1$ or $stmt_2$ is executed depending on the result of the test. Since we want to estimate an upper bound on the execution time statically, we simply take the maximum of the estimated execution times for $stmt_1$ and $stmt_2$. As an example, consider the code

$x := y + 1;$
`if` $(x > z)$ `then` $y := z$ `else` $\{y := 0;\ z := x + 1\}$.

Suppose the execution time of each of the assignment statements is c_1, and the execution time of evaluating the condition $(x > z)$ is c_2, then the execution time of the above code is estimated to be $3c_1 + c_2$.

Thus, a WCET bound for straight-line code can be obtained from execution times for assignment statements and for evaluating Boolean expressions. Let us consider the assignment statement $x := y + 1$. The time it takes to execute such an instruction depends on the specifics of the underlying architecture and, particularly, how memory is organized. If the variables x and y are stored in registers, then such an assignment maps to a single machine instruction and executes in one clock cycle. In contrast, if the variables x and y reside in main memory, then executing such an assignment requires fetching the value of y from the main memory, incrementing a register, and then storing it back in the memory. When an assignment involves such memory operations, its execution time varies over a wide range depending on whether each relevant memory address resides in the local cache or whether it resides in the main memory. If we assume every read or write results in a cache-miss and leads to an access of the main memory, then the resulting upper bound is likely to be too pessimistic for useful analysis. In particular, for the code above, the variables x and y are accessed multiple times, and only the first such access can lead to a cache-miss. As this example illustrates, estimating an upper bound on the execution time that (1) is not too pessimistic, (2) reflects the complexity of the underlying architecture, (3) is guaranteed to be an upper bound on the actual execution time in all cases, and (4) can be computed by statically analyzing the code with reasonable computational effort is a challenging problem.

Periodic Job Model

A periodic job model then consists of a set of periodic jobs, where each job is specified using a period, a deadline, and a bound on execution time. This definition is summarized below.

> Periodic job model
>
> A periodic job model consists of a finite set $\mathcal{J}$ of jobs, where each job J has an associated period $\pi(J)$, deadline $\delta(J)$, and worst-case execution time $\eta(J)$, each of which is a positive integer, such that the condition $\eta(J) \leq \delta(J) \leq \pi(J)$ holds.

As an example, consider the job model consisting of two periodic jobs J_1 and J_2: the job J_1 has period 5, deadline 4, and WCET 3; and the job J_2 has period 3, deadline 3, and WCET 1. The scheduling problem then is to allocate computation time to these two jobs so that for every $a \geq 0$, the job J_1 gets three time units between times $5a$ and $5a+4$, and the job J_2 gets one time unit between times $3a$ and $3a+3$.

Exercise 8.1: Consider the code

$x := y + 1;$
`if` $(x > z)$ `then` { `if` $y > 1$ `then` $y := z$} `else` $\{y := 0;\ z := x + 1\}$.

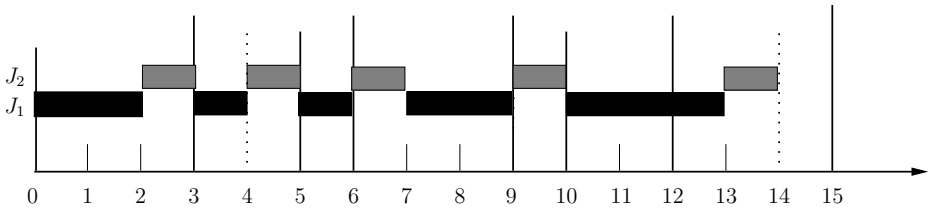

Figure 8.2: Illustrative Schedule for a Job Model with Two Jobs

Assuming that each atomic assignment statement takes c_1 time units and evaluation of each Boolean expression used in the conditional tests takes c_2 time units, estimate the worst-case execution time of the above code. ■

8.1.3 Schedulability

A schedule specifies allocation of processing time to jobs, and the scheduling problem is to find a schedule that meets the deadlines of all the jobs.

Schedules and Feasibility

As discussed earlier, the scheduler allocates processing time in discrete time slots. A schedule σ for a periodic job model $\mathcal{J}$ specifies for every time t, for $t = 0, 1, 2, \ldots$, the job $J \in \mathcal{J}$ that is allocated the time slot starting at time t. It is possible that a particular time slot is allocated to none of these jobs, and we use $\perp$ to indicate this possibility. Formally, a schedule σ is a function from the set `nat` of natural numbers to the set $\mathcal{J} \cup \{\perp\}$: $\sigma(t) = J$ means that the time slot starting at time t is allocated to the job J, and $\sigma(t) = \perp$ means that the time slot starting at time t is not allocated to any of the jobs in the set $\mathcal{J}$. When $\sigma(t) = \perp$, the processor can stay idle during this time slot or can be used for computation not related to the system corresponding to this job model.

In this scheduling framework, the three parameters $\eta(J)$, $\delta(J)$, and $\pi(J)$ corresponding to a periodic job J specify the requirement that, for every instance a, the job J should be allocated a total of $\eta(J)$ time slots in the interval from time $\alpha(J, a)$ to time $\delta(J, a)$. A schedule σ is *deadline-compliant* for a job J if it indeed allocates the necessary number of slots to each instance of this job. A schedule is deadline-compliant for a periodic job model if it is deadline-compliant for each of the jobs in the model.

Let us reconsider the job model consisting of two periodic jobs J_1 and J_2: the job J_1 has period 5, deadline 4, and WCET 3; and the job J_2 has period 3, deadline 3, and WCET 1. Consider the schedule whose allocation pattern for the first 15 slots is shown in figure 8.2. For each job, vertical lines indicate times when successive instances of the job are ready for execution, and dashed vertical lines indicate the corresponding deadlines (for the job J_2, the deadline coincides

with the period, so these dashed lines are missing). When a slot is assigned to a job, the row corresponding to that job is shown as a filled rectangle. Thus, in the schedule of figure 8.2, the first two slots are assigned to the first job, the third slot is assigned to the second job, and the 15th slot is not assigned to any of the two jobs. The same pattern repeats for every 15 slots. Formally, the schedule σ is given by:

$$\sigma(0) = \sigma(1) = \sigma(3) = \sigma(5) = \sigma(7) = \sigma(8) = \sigma(10) = \sigma(11) = \sigma(12) = J_1;$$
$$\sigma(2) = \sigma(4) = \sigma(6) = \sigma(9) = \sigma(13) = J_2;$$
$$\sigma(14) = \perp;$$
$$\text{for every } t \geq 0,\ \sigma(t + 15) = \sigma(t).$$

The abstraction of a schedule as an assignment of time slots to jobs can naturally be implemented in the scheduler architecture of figure 8.1. For instance, to implement the schedule shown in figure 8.2, the scheduler transmits the following events to the job J_1: run_1 at time 0, $preempt_1$ at time 2, run_1 at time 3, run_1 at time 5, $preempt_1$ at time 6, run_1 at time 7, and run_1 at time 10.

The illustration in figure 8.2 should convince you that this schedule is deadline-compliant for both jobs: each instance of the job J_1 is assigned three slots within four time units of its arrival time, and each instance of the job J_2 is assigned one slot before its next instance arrives.

For a periodic job model $\mathcal{J}$, if there exists a deadline-compliant schedule, then the job model is called *schedulable*. Thus, the model consisting of the job J_1 with period 5, deadline 4, and WCET 3 and the job J_2 with period 3, deadline 3, and WCET 1 is schedulable. Now suppose we change the WCET of the job J_1 to 4. Then convince yourself that the job model is not schedulable: during the first 10 slots, the deadlines of at least the first two instances of the job J_1 must be met, and thus it must be given at least eight slots; at the same time, the deadlines of at least the first three instances of the job J_2 must be met, implying that it must be given at least three slots, which is not possible.

These definitions are summarized below. For notational convenience, for a schedule σ, a job J, and time instances t_1 and t_2, such that $t_1 < t_2$, let us denote the number of time slots that the schedule σ allocates to the job J between times t_1 and t_2 by $\sigma(t_1, t_2, J)$, that is,

$$\sigma(t_1, t_2, J) \ = \ |\{t \mid t_1 \leq t < t_2 \text{ and } \sigma(t) = J\}|.$$

Schedulability of Periodic Job Model

A schedule σ for a periodic job model $\mathcal{J}$ is a function that maps every natural number $t \geq 0$ to the set $\mathcal{J} \cup \{\perp\}$. Such a schedule is deadline-compliant for a job $J \in \mathcal{J}$ if for every instance $a \geq 1$, $\sigma(\alpha(J, a), \delta(J, a), J) = \eta(J)$. The schedule σ is deadline-compliant for the job model $\mathcal{J}$ if it is deadline-compliant for every job in $\mathcal{J}$. The job model $\mathcal{J}$ is *schedulable* if there exists a deadline-compliant schedule σ for $\mathcal{J}$.

Periodic Schedules

A periodic schedule assigns slots to jobs in a repeating manner. Formally, a schedule σ is a *periodic schedule* with period p, where p is a positive number, if for all time instances $t \geq 0$, $\sigma(t+p)$ equals $\sigma(t)$. The schedule shown in figure 8.2 is a periodic schedule with period 15. A periodic schedule σ can be fully specified by listing its period p and the assignments $\sigma(0), \sigma(1), \ldots \sigma(p-1)$ of slots to jobs for the first p slots.

If we want to check whether a periodic job model is schedulable, then we can limit the search for plausible schedules to only periodic schedules. This is established in the following theorem. Its proof shows that given a periodic job model, it suffices to consider periodic schedules whose period equals the least-common multiple of the periods of all the jobs in the model.

Theorem 8.1 [Periodic Schedules] *A periodic job model $\mathcal{J}$ is schedulable if and only if there exists a periodic schedule that is deadline-compliant for $\mathcal{J}$.*

Proof. Consider a periodic job model $\mathcal{J}$. If there exists a periodic schedule that is deadline-compliant for all the jobs in $\mathcal{J}$, then by definition the job model $\mathcal{J}$ is schedulable. Conversely, suppose the job model $\mathcal{J}$ is schedulable. Then by definition there exists a deadline-compliant schedule σ. The schedule σ need not be periodic, and our goal is to construct a *periodic* schedule σ' that also meets the deadlines of all the jobs.

Let p be the least-common multiple of periods of all the jobs, that is, of all the numbers in the set $\{\pi(J) \mid J \in \mathcal{J}\}$. Define the desired schedule σ' as follows: for $0 \leq t < p$, $\sigma'(t) = \sigma(t)$, and for every $t \geq 0$, $\sigma'(t+p) = \sigma'(t)$. Clearly, the schedule σ' is periodic with period p. Consider a job $J \in \mathcal{J}$. Let $n = p/\pi(J)$ (note that, by the choice of p, it is divisible by $\pi(J)$). Since the schedule σ' is the same as the schedule σ for the first p slots and the schedule σ is deadline-compliant for the job J, the schedule σ' also meets the deadlines of the first n instances of the job J. Since the schedule σ' is periodic, for every $a \geq 1$, the quantity $\sigma(\alpha(J,a), \delta(J,a), J)$ equals the quantity $\sigma(\alpha(J,a+n), \delta(J,a+n), J)$, and thus if it meets the deadline of the instance J^a, then it also meets the deadline of the instance J^{a+n}. This shows that the schedule σ' is deadline-compliant for the job model. ■

Utilization

Consider the periodic job model consisting of two jobs, the job J_1 with period 5, deadline 4, and WCET 3; and the job J_2 with period 3, deadline 3, and WCET 1. Since the job J_1 requires three time slots in every five slots, it needs 3/5 of the available processing time. Similarly, since the job J_2 requires one time slot in every three slots, it needs 1/3 of the available processing time. The sum $3/5 + 1/3$, equal to 14/15, is called the *utilization* of the job model. Formally,

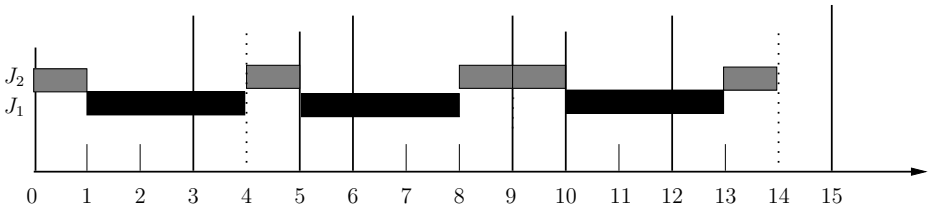

Figure 8.3: A Non-preemptive Schedule for a Job Model with Two Jobs

the utilization of a periodic job model $\mathcal{J}$ is defined as

$$U(\mathcal{J}) \;=\; \sum_{J \in \mathcal{J}} \eta(J)/\pi(J).$$

The utilization indicates what fraction of the available processing time is necessary to meet the demands of all the jobs. Note that the periodic schedule of figure 8.2 for the job model in our example assigns 14 out of 15 slots for scheduling of the two jobs, leaving one slot unassigned.

If the utilization of a job model exceeds 1, then it means that all the jobs together need more processing time than is available. In such a case, the job model is not schedulable. In our example job model, if we change the WCET of the job J_1 to 4, then utilization changes to $4/5 + 1/3$, which exceeds 1, and, as already noted, this leads to non-schedulability. Since the utilization of a job model can be computed easily, checking whether it exceeds 1 is a quick test to detect non-schedulability.

Preemptive vs. Non-preemptive Schedules

Consider the schedule of figure 8.2. The first instance of the job J_1 needs three out of the first four slots, and it is chosen at times 0, 1, and 3. While this meets the deadline of this instance, at time 2, its execution must be interrupted (using the event $preempt_1$) to let the job J_2 execute during the next slot and needs to be resumed in the subsequent slot. The schedule of figure 8.2 is called a preemptive schedule due to its use of such preemptions. A schedule that does not include such preemptions is called non-preemptive. In other words, a non-preemptive schedule assigns each instance of a job J, a chunk of $\eta(J)$ many *consecutive* time slots. Formally, a schedule σ over a set $\mathcal{J}$ of jobs is preemptive if there exists a job J and times $t_1 < t_2 < t_3$, such that (1) $\sigma(t_1) = \sigma(t_3) = J$ and $\sigma(t_2) \neq J$, and (2) there exists a such that $\alpha(J, a) \leq t_1$ and $t_3 < \alpha(J, a + 1)$. This says that there exist three slots, all belonging to an interval during which a single instance of the job J is active, such that the job J is assigned the two extreme slots but not the middle one.

Since preemption of jobs requires switching the context of processing from one job to another, it is expensive to implement and should be avoided whenever

possible. For our example job model consisting of the job J_1 with period 5, deadline 4, and WCET 3; and the job J_2 with period 3, deadline 3, and WCET 1, figure 8.3 shows an alternative periodic schedule that is non-preemptive and yet is deadline-compliant.

It is possible that the only way to meet all the deadlines is by relying on preemption: a periodic job model can be schedulable, and yet there may be no non-preemptive schedule that is deadline-compliant. To illustrate this, suppose the model consists of two jobs: the job J_1 with period 2, deadline 1, and WCET 1; and the job J_2 with period 4, deadline 4, and WCET 2. The periodic preemptive schedule, with period 4, that chooses the job J_1 at times 0 and 2, and chooses the job J_2 at times 1 and 3, is deadline-compliant. But it is easy to convince yourself that it is not possible to find a non-preemptive schedule that meets all the deadlines.

Scheduling Policies

Given a periodic job model $\mathcal{J}$, the goal of a scheduling policy is to either produce a deadline-compliant periodic schedule or report failure to find such a schedule. While we will discuss two such policies in detail in sections 8.2 and 8.3, let us consider some of the questions we should ask to understand and evaluate a scheduling policy.

When does the policy succeed in producing a schedule? Ideally, a policy should produce a deadline-compliant periodic schedule whenever the job model is schedulable. If this is not the case, then we should aim to find conditions under which the policy is guaranteed to succeed.

How much computational effort does the policy need to compute the schedule? We already know that if the job model is schedulable, then there exists a periodic deadline-compliant schedule with period equal to the least-common multiple of the periods of all the jobs. If the model has n jobs, then each slot can be assigned only $(n + 1)$ different values (since a schedule maps a slot to either one of the jobs or to $\perp$). Thus, there are p^{n+1} many different periodic schedules possible with period p. A naive scheduling policy analyzes all such possible schedules and chooses a schedule that meets all the deadlines. However, such a policy is too inefficient as its computational cost grows rapidly as the number of jobs and the values of the periods grow. A scheduling policy is required to be *efficient* with time-complexity polynomial in the number of jobs in the model.

How much of the decision logic of the policy is implemented off-line vs. on-line? One possible way to implement a scheduling policy for periodic job models is to compute the desired schedule off-line, that is, before the system starts executing. Then during the execution of the system, that is, on-line, the scheduler simply needs to look up this schedule at the beginning of each time slot (or whenever a scheduling decision regarding allocation of jobs to the processor needs to be made). Alternatively, some scheduling policies only assign priorities to jobs off-line, and when a scheduling decision needs to be made on-line, make the

decision based on, say, comparing job priorities. Policies of this latter kind are preferable as there is no need to compute and store a long schedule, and such policies can typically be extended to more complex job models, for instance, to job models in which jobs are added and removed dynamically as the system executes. However, when the decisions are made on-line, it is critical that the overhead associated with such a decision is minimal–no more than a couple of instructions.

Does the policy ensure alternative optimality criteria? Our definition of schedulability requires that the schedule should meet all the deadlines. When there are multiple deadline-compliant schedules, alternative criteria can be used to prefer one schedule over the other. For example, we may want the scheduling policy to compute a schedule with the least number of preemptions (and produce a non-preemptive schedule whenever possible). Another such criterion is *response time*: the response time of an instance of a job is the difference between the arrival time of this instance and the time this instance finishes its execution according to the schedule. We may want the scheduling policy to compute a schedule that minimizes the average response time over all the job instances.

Exercise 8.2: Consider a periodic job model with two jobs: the job J_1 has period 5, deadline 5, and WCET 2; and the job J_2 has period 7, deadline 7, and WCET 4. What is the utilization for this job model? Show a periodic deadline-compliant schedule. ■

Exercise 8.3: Consider a periodic job model with two jobs: the job J_1 has period 3, deadline 2, and WCET 1; and the job J_2 has period 5, deadline 5, and WCET 3. Argue that this job set is schedulable only if we allow preemptions. Find a deadline-compliant schedule with minimum number of preemptions. ■

8.1.4 Alternative Job Models

The periodic job model is the simplest model for formalizing the demand for processing time by jobs in a real-time application. Many extensions and variations have been studied in the literature. We close this section with a brief introduction to some of the most significant variants.

Precedence Constraints among Jobs

In a periodic job model with precedence constraints, in addition to the period, deadline, and WCET for each job, precedence constraints among jobs are also specified. The precedence constraint $J_1 \prec J_2$ between two jobs J_1 and J_2 means that for every a, the ath instance of the job J_2 should start executing only after the ath instance of the job J_1 has finished its execution. It is required that whenever there is such a precedence constraint between two jobs, they should have the same period, and the precedence relation should be acyclic. The schedulability problem then is to find a schedule that not only meets the deadlines of all the jobs but also obeys the ordering constraints expressed by

the precedence relation. Precedence constraints among tasks in the task-graph-based description of synchronous components in chapter 2 naturally lead to a job model with precedence constraints.

Dynamically Changing Job Set

In the basic periodic job model, it is assumed that the set of jobs is known *a priori* and is fixed. In practice, it is desirable to allow a *dynamically changing* set of jobs, where jobs can be added and removed while the system is executing. In such a scenario, whenever a new job is to be added, the scheduling policy needs to perform a schedulability test on the revised set of jobs and admit the new job only when the schedulability test is successful. The policy also needs to make scheduling decisions dynamically at run-time since the schedule cannot be computed off-line in advance.

Aperiodic Jobs

An *aperiodic* job is a job whose arrival pattern is irregular and not known in advance. For example, in the cruise controller design discussed in chapter 2, while the tasks corresponding to measuring the current speed and controlling the speed (the components `MeasureSpeed` and `ControlSpeed` of figure 2.29) are to be executed in a periodic manner, the task responsible for updating the cruising speed (the component `SetSpeed`) needs to be executed whenever the driver switches on the cruise control or decides to change the cruising speed. An aperiodic job, thus, has an associated deadline and worst-case execution time, but instead of a period, it has an associated triggering event. When the job set has both periodic and aperiodic jobs, the scheduling policy is inspired by the principles and analysis used for design of policies for purely periodic job models and sets aside a fraction of the time slots for allocation to the anticipated but unpredictable demands by the arrival of aperiodic jobs. Such a policy, however, cannot offer guaranteed deadline-compliance, and we can only hope for a "best-effort" policy, where the guarantees of the policy are measured only by comparing them with respect to the guarantees of alternative policies.

Multiprocessor Scheduling

Our definition of a schedule allocates each time slot to at most one job, and this corresponds to the assumption that all the jobs are executing on a single processor. Modern computing platforms consist of multiple computing cores, and in embedded applications, it is even more common to have specialized processors dedicated to executing specific jobs. To formalize the problem of scheduling jobs on multiprocessors, in addition to the period, deadline, and worst-case execution time for each job, we also specify the set of processors, and for each processor, the subset of jobs that can execute on this processor. The multiprocessor schedule then maps each time slot and each processor to a job (or $\perp$ to indicate an idle slot), and the scheduling problem is to compute a deadline-compliant multiprocessor schedule. A job set that is not schedulable on a single processor can

become schedulable on multiple processors due to the availability of additional processing time. However, designing an efficient scheduling policy to compute a deadline-compliant schedule in the multiprocessor job model is more challenging since the multiprocessor schedulability problem is computationally intractable (typically NP-complete).

Soft vs. Hard Real-Time Requirements

In the scheduling problem we have defined, a schedule is required to meet the deadlines of all the instances of each job. While such strict deadline-compliance is a necessary requirement in safety-critical and real-time control systems, in applications such as multimedia, it may be appropriate to demand a weaker guarantee. For example, if the job corresponds to refreshing the screen by displaying the next video frame, executing only 95% of all instances of the job may be acceptable. In the literature, systems where it is required that all deadlines must be met, as missing deadlines implies an unacceptable safety violation, are called *hard real-time* systems, and systems where deadlines should be met as frequently as possible, but missing deadlines only causes a degradation of the desired quality, are called *soft real-time* systems. The objective of a scheduling policy for soft real-time systems is then to compute a schedule with a minimum fraction of missed deadlines.

Exercise 8.4*: Consider a set $\mathcal{J}$ of n jobs with a precedence relation $\prec$ such that $\prec$ is acyclic. Suppose every job in $\mathcal{J}$ has period p, deadline p, and WCET c. Let us assume that we have n processors available for scheduling, and each job can be executed on any of the processors. This job set is schedulable if we can find a multiprocessor schedule that meets the precedence constraints (that is, if $J_1 \prec J_2$, then in each period, the job J_1 should finish executing before the job J_2 can start executing, but it's okay if they execute on different processors). Under what conditions is the job set schedulable? Hint: the condition should relate period p, WCET c, and some quantity derived from the precedence relation $\prec$. ■

8.2 EDF Scheduling

The Earliest Deadline First (EDF) policy is a classical scheduling policy that always selects a job whose deadline is going to expire first. This scheduling policy is applicable to a wide range of job models and is commonly used in practice. We will first describe the EDF policy for the periodic job model and then analyze its performance.

8.2.1 EDF for Periodic Job Model

Given a set of periodic jobs, at every time t, the EDF scheduling policy assigns the next slot to the job that has the earliest (or least) deadline. In a periodic model, for each job, different instances of the job are active at different times,

and thus the specific value of the deadline for a job depends on the time t. Also, the policy assigns a slot to a job only if the demand of its active instance has not already been met. The construction of the schedule according to these rules is formalized below.

Scheduling Policy

Consider a periodic job model $\mathcal{J}$, where each job J has an associated period $\pi(J)$, deadline $\delta(J)$, and worst-case execution time $\eta(J)$. For each time $t = 0, 1, 2, \ldots$, the EDF scheduling policy decides allocation of the next slot and builds the EDF schedule σ step by step in the following manner. Consider a job J. Let a be the unique number such that $\alpha(J, a) \leq t < \alpha(J, a + 1)$. The deadline of the job J at time t is the deadline of this instance of J, namely, $\delta(J, a)$. If the schedule σ up to the first t slots has already allocated $\eta(J)$ number of slots to this particular instance of the job J, then it does not need any more processing time. To formalize this, we say that the job J is *ready* at time t according to the schedule σ if $\sigma(\alpha(J, a), t, J) < \eta(J)$. If there is no job that is ready at time t, then the next slot is left unassigned, that is, $\sigma(t) = \perp$. Otherwise it selects a job J such that the job J is ready at time t and has the least deadline at time t among the jobs that are ready at time t, that is, $\sigma(t) = J$, such that the job J is ready at time t and if there is another job K that is also ready at time t, then the deadline of the job J at time t is less than or equal to the deadline of the job K at time t. Note that if multiple ready jobs share the same deadline, then any one of them can be chosen according to the EDF policy, and the choice is made using some alternative criteria in a specific implementation.

The definition of an EDF schedule is summarized below.

> EDF Schedule
>
> A schedule σ for a periodic job model $\mathcal{J}$ is called an *EDF-schedule* if for every time $t \geq 0$, if no job is ready at time t in the schedule σ, then $\sigma(t) = \perp$, or else $\sigma(t) = J$, such that the job J is ready at time t in the schedule σ, and for every job $K \in \mathcal{J}$, either the job K is not ready at time t or the deadline of the job J at time t is less than or equal to the deadline of the job K at time t.

Example

Let us revisit the periodic job model consisting of the job J_1 with period 5, deadline 4, and WCET 3; and the job J_2 with period 3, deadline 3, and WCET 1 (see deadline-compliant periodic schedules of figures 8.2 and 8.3 for this model). Suppose we want to construct a schedule according to the EDF policy for this model.

Initially, at time $t = 0$, both jobs J_1 and J_2 are ready (since the demands of their respective first instances have not yet been met), the deadline for the job J_1 is 4, and the deadline for the job J_2 is 3. The EDF policy hence assigns the first slot

to the job J_2. As a result, at time 1 as well as at time 2, the job J_2 is no longer ready (the demand of the corresponding active instance has been met), and the policy picks the job J_1 at these times. At time 3, the second instance of the job J_2 arrives, and thus both jobs are ready at time 3. At this point, the deadline for the job J_1 is still 4, but the deadline for the job J_2 is now 6. As a result, the EDF policy assigns the next time slot to the job J_1. At time 4, the job J_1 is no longer ready, and hence the subsequent slot is assigned to the job J_2. At time 5, the job J_2 is no longer ready as the demand for its active instance has been met, but the job J_1 is ready again as its second instance arrives. Thus, the policy chooses the job J_1 at time 5. At time 6, the third instance of the job J_2 arrives, and both jobs are ready. At this time, the deadlines for both jobs equal 9. As a result, the EDF policy is free to choose either of the two jobs. Suppose it chooses the job J_1 since its identifier is smaller than that of the job J_2. By the same reasoning, the next slot is also allocated to the job J_1. At times 8 and 9, only the job J_2 is ready and gets chosen. At times 10 and 11, only the job J_1 is ready, and gets chosen. At time 12, both jobs are ready, the deadline for the job J_1 is 14, and for the job J_2 is 15. Thus, the policy allocates the next slot to the job J_1. At time 13, only the job J_2 is ready and gets the next slot. At time 14, none of the jobs is ready, and hence the following slot is unassigned. The same cycle now repeats with a period of 15. The EDF schedule constructed in this manner is, in fact, identical to the schedule shown in figure 8.3.

Properties of the EDF Policy

In section 8.2.2, we will study under what conditions the EDF policy is guaranteed to produce a deadline-compliant schedule. For now, let us note some of its basic properties.

Whenever a scheduling decision is to be made, the EDF policy picks the job with the earliest current deadline among the ready jobs without explicitly analyzing the global consequences of such a decision. Such a policy is an example of a class of algorithms known as *greedy* algorithms.

Let us assume that whenever the EDF policy needs to choose a job among the ready jobs with identical deadlines, it uses some fixed decision rule (for example, choose the job with the lowest identifier). With this assumption, observe that the schedule produced by the EDF policy is a *periodic* schedule with period equal to the least-common multiple of periods of all the jobs.

In the EDF schedule of figure 8.3, at time 0, the job J_2 is given a priority over the job J_1, while at time 3, the job J_1 is given a priority over the job J_2. Such a scheduling policy whose relative preference among the ready jobs is different at different times is called a *dynamic priority* policy.

In general, the schedule generated by the EDF policy can be *preemptive*. For example, consider the periodic job model with the job J_1 with period 2, deadline 1, and WCET 1; and the job J_2 with period 4, deadline 4, and WCET 2. The

EDF policy chooses the job J_1 at times 0 and 2 and chooses the job J_2 at times 1 and 3. The resulting schedule is deadline-compliant and is preemptive.

In our description of the EDF policy, the policy makes a decision regarding which job is to be scheduled in every slot. However, the decision logic need not be executed in every slot based on the following observation. If a job J is chosen at time t, then the EDF policy is guaranteed to choose the same job at time $t+1$ also if both the following conditions hold: (1) the execution of the currently active instance of the job J is not yet finished, and (2) no new instance of any other job K arrives at time $t+1$. For example, in the schedule of figure 8.3, the job J_1 is chosen at time 1, and at the next time instance, neither the current job finishes its execution nor a new instance of the job J_2 arrives, and this ensures that the job J_1 is chosen at time 2 also. In other words, the scheduler needs to make a decision about allocation of processing time to jobs only when the set of ready jobs changes, which can occur only when either the current instance of the job finishes its execution or a new instance of another job arrives.

This suggests a natural strategy for implementing the EDF policy in the scheduler architecture of figure 8.1. The scheduler maintains a list *WaitingJobs*, which contains all the jobs that are waiting for processing time in the decreasing order of their relative priorities. When the job that is currently running finishes its execution, if the list *WaitingJobs* is non-empty, then the first job from the list *WaitingJobs* is removed and is allocated the processor. When a new instance of a job J arrives, the scheduler performs the following steps. If no job is currently assigned the processor, then the job J is allocated processing time. Otherwise, suppose the job J' is currently assigned the processor. If the deadline of the newly arrived instance of J is less than the deadline of the job J', then the job J' is preempted and added to the front of the queue *WaitingJobs*, and the job J is allocated the processor. If not, the newly arrived instance of the job J is inserted in the queue *WaitingJobs* of ready jobs in the suitable position by comparing its deadline with those of the jobs already in this sorted list.

Exercise 8.5: For the periodic job model consisting of the job J_1 with period 5, deadline 4, and WCET 3; and the job J_2 with period 3, deadline 3, and WCET 1, figure 8.3 shows the schedule constructed by the EDF policy assuming that whenever both jobs are ready with identical deadlines, the job J_1 is chosen. Show the schedule constructed by the EDF policy assuming that whenever both jobs are ready with identical deadlines, the job J_2 is chosen. ■

Exercise 8.6: Consider a periodic job model consisting of the job J_1 with period 6, deadline 5, and WCET 2; the job J_2 with period 8, deadline 4, and WCET 2; and the job J_3 with period 12, deadline 8, and WCET 4. Construct the schedule according to the EDF policy (if multiple ready jobs have identical deadlines, prefer the job J_1 over the job J_2 over the job J_3). ■

8.2.2 Optimality of EDF

Given a periodic job model, when is the EDF scheduling policy guaranteed to produce a deadline-compliant schedule? As the next theorem shows, the EDF policy is guaranteed to succeed as long as the model is schedulable. That's why the EDF policy is considered to be an *optimal* algorithm: it is guaranteed to produce a deadline-compliant schedule as long as one such schedule exists.

Theorem 8.2 [Optimality of EDF] *If $\mathcal{J}$ is a schedulable periodic job model and σ is an EDF schedule for $\mathcal{J}$, then σ is deadline-compliant.*

Proof. Let $\mathcal{J}$ be a schedulable periodic job model, and let σ be an EDF schedule for $\mathcal{J}$. We want to prove that the schedule σ meets the deadlines of all instances of all jobs. The proof is by contradiction. That is, we assume that the schedule σ misses some deadline, and we will arrive at a contradiction. Suppose there is an instance of a job such that the schedule σ does not allocate this instance enough slots by its deadline, and let t_0 be the time of this missed deadline.

Since the job model $\mathcal{J}$ is schedulable, there exists a schedule, say σ', that meets all the deadlines. Consider the two schedules σ and σ'. Since the schedule σ' meets all the deadlines and the schedule σ misses at least one deadline, the two schedules cannot be identical. Let $\mathit{diff}(\sigma, \sigma')$ denote the first time instance where the two schedules make different choices, that is, $\mathit{diff}(\sigma, \sigma') = t$, such that $\sigma(t) \neq \sigma'(t)$ and for all $t' < t$, $\sigma(t') = \sigma'(t')$. Such a time instance t must be less than the deadline t_0 that the schedule σ misses but the schedule σ' does not miss.

There can be multiple deadline-compliant schedules for the job model $\mathcal{J}$. Let σ_1 be a schedule among all such deadline-compliant schedules σ', such that $\mathit{diff}(\sigma, \sigma')$ is the largest, that is, the schedule σ_1 is a deadline-compliant schedule for $\mathcal{J}$, such that if the schedule σ' is also a deadline-compliant schedule for $\mathcal{J}$, then $\mathit{diff}(\sigma, \sigma') \leq \mathit{diff}(\sigma, \sigma_1)$. In other words, the schedule σ_1 makes the same choices as the EDF schedule σ for as long as possible without giving up the goal of staying deadline-compliant.

Let $t_1 = \mathit{diff}(\sigma, \sigma_1)$. We know that $\sigma(t_1) \neq \sigma_1(t_1)$, and for all $t < t_1$, $\sigma(t) = \sigma_1(t)$. We will consider different cases based on the values of $\sigma(t_1)$ and $\sigma_1(t_1)$. In each case, we construct another schedule σ_2 such that (1) the schedule σ_2 is deadline-compliant, and (2) $\mathit{diff}(\sigma, \sigma_2) > t_1$. This is a contradiction to the way the schedule σ_1 is chosen, implying that our initial assumption that the schedule σ misses some deadline cannot be true, thus completing the proof.

Consider the case when $\sigma(t_1) = J$ and $\sigma_1(t_1) = K$, with $J \neq K$, such that the job K is ready at time t_1 (see figure 8.4 for illustration). Let the deadlines of the jobs J and K at time t_1 be t_J and t_K, respectively. Since the EDF schedule chooses the job J at time t_1, the job J must be the one with the earliest deadline among all the jobs that are ready at time t_1, and this implies $t_J \leq t_K$. Since

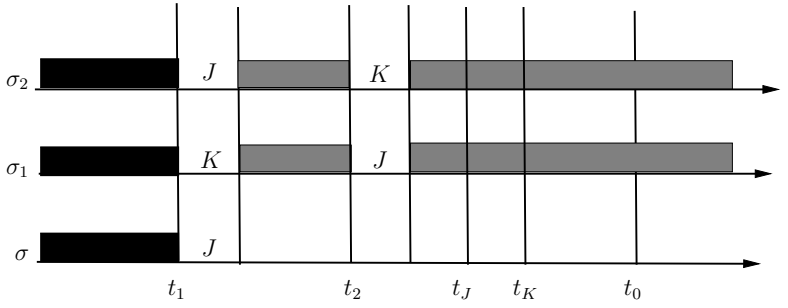

Figure 8.4: Proof of Optimality of EDF

the job J is ready at time t_1, the demand of its active instance at time t_1 has not yet been met, and it needs at least one more slot before the deadline t_J. Since the schedule σ_1 is deadline-compliant, there must be a time instance t_2 such that $t_1 < t_2 < t_J$ and $\sigma_1(t_2) = J$.

Now define the schedule σ_2 such that for all $t \neq t_1$ and $t \neq t_2$, $\sigma_2(t) = \sigma_1(t)$ and $\sigma_2(t_1) = J$ and $\sigma_2(t_2) = K$ (see figure 8.4 for illustration). That is, the schedule σ_2 is obtained from the deadline-compliant schedule σ_1 by swapping the choice of the jobs J and K at times t_1 and t_2. By the choice of these times, the instances t_1 and t_2 belong to the same active instance of the job J and the same instance of the job K, both before their relevant deadlines. As a result, the number of slots allocated to each instance of each job is exactly the same in both the schedules σ_1 and σ_2. It follows that the schedule σ_2 is also deadline-compliant. Now, the EDF schedules σ and σ_2 are identical for times less than t_1 and also at time t_1, and thus the condition $\mathit{diff}(\sigma, \sigma_2) > t_1$ must hold.

For the remaining cases, either (1) $\sigma(t_1) = \perp$ and $\sigma_1(t_1) \neq \perp$, (2) $\sigma(t_1) = J$ and $\sigma_1(t_1) = \perp$, or (3) $\sigma(t_1) = J$ and $\sigma_1(t_1) = K$, such that the job K is not ready at time t_1. In all these cases, define the schedule σ_2 such that for all $t \neq t_1$, $\sigma_2(t) = \sigma_1(t)$ and $\sigma_2(t_1) = \sigma(t_1)$. We leave it as an exercise to verify that the resulting schedule σ_2 is deadline-compliant and the condition $\mathit{diff}(\sigma, \sigma_2) > t_1$ must hold. ■

The proof shows that, at any step, if a deadline-compliant schedule chooses a job K over another job J, whose deadline is earlier than that of K, then choosing the job J, instead of the job K, cannot be the cause of a missed deadline. This core idea of the proof applies to more general job models and shows, for instance, even in the presence of both periodic and aperiodic jobs, that the EDF policy produces a deadline-compliant schedule as long as one exists.

Exercise 8.7: Complete the proof of theorem 8.2: show that the schedule σ_2 defined in the last paragraph of the proof is such that it meets all the deadlines and the condition $\mathit{diff}(\sigma, \sigma_2) > t_1$ holds. ■

Exercise 8.8: We have established that if there exists a deadline-compliant schedule, then the EDF policy produces one such schedule. Show that, however, the following statement is false: if there exists a deadline-compliant schedule with no preemptions, then the EDF policy is guaranteed to produce one such schedule. That is, construct a job model $\mathcal{J}$ (with two jobs) such that (1) there exists a deadline-compliant schedule σ that has no preemptions, and (2) the deadline-compliant schedule produced by the EDF policy involves preemptions. ■

8.2.3 Utilization-Based Schedulability Test

We know that as long as the periodic job model is schedulable, the EDF policy produces a deadline-compliant schedule. But can we test whether the periodic job model is schedulable or, equivalently, whether the EDF policy is going to succeed without explicitly generating an EDF schedule and checking whether it meets all the deadlines. It turns out that when all the deadlines are implicit, that is, for every job J, the deadline $\delta(J)$ equals its period $\pi(J)$, the test for schedulability is simple: a periodic job model is schedulable exactly when its utilization is 1 or less. We already know that if the utilization exceeds 1, the demand for processing time exceeds the available processing time, and thus the job model is not schedulable. The following theorem proves that if the utilization is 1 or less, then an EDF schedule is deadline-compliant.

Theorem 8.3 [Schedulability test when deadlines equal periods] *Let $\mathcal{J}$ be a periodic job model such that for every job J, $\delta(J) = \pi(J)$. Then the job model $\mathcal{J}$ is schedulable if and only if $U(\mathcal{J}) \leq 1$.*

Proof. Let $\mathcal{J}$ be a periodic job model such that for every job K, $\delta(K) = \pi(K)$. Let

$$U = \sum_{K \in \mathcal{J}} \eta(K)/\pi(K) \qquad (1)$$

be the utilization of the job model.

If $U > 1$, then the total demand for the processing time exceeds the available supply, and there does not exist a deadline-compliant schedule, and the job model is not schedulable.

For the converse, assume that the job model $\mathcal{J}$ is not schedulable. Consider an EDF schedule σ for $\mathcal{J}$. The schedule σ cannot be deadline-compliant. We proceed to prove that the utilization U must exceed 1.

Since σ is not deadline-compliant, there must exist a job J and an instance i of the job J, such that the schedule σ does not allocate enough time slots to the instance J^i. Let t_1 be the arrival time of this instance, that is, $t_1 = \alpha(J, i)$, and let t_2 be the deadline of this instance, that is, $t_2 = \delta(J, i)$, which is the same as $\alpha(J, i+1)$ (see figure 8.5). We know that the instance J^i misses its deadline,

and thus the schedule σ chooses the job J for fewer than $\eta(J)$ time instances in the interval $[t_1, t_2)$: the condition $\sigma(t_1, t_2, J) < \eta(J)$ must hold.

Consider a time instance t with $t_1 \leq t < t_2$. We know that the job J is ready at time t and its deadline is t_2. The EDF schedule σ at time t must choose some job keeping the processor busy: $\sigma(t) \neq \perp$. Furthermore, the EDF policy selects a job with the earliest deadline, and hence if $\sigma(t) = K$, then the deadline of the job K at time t must be t_2 or less.

Let t_0 be the least time such that for all time instances t with $t_0 \leq t < t_2$, the schedule σ chooses some job K at time t such that the deadline of the job K at time t is $\leq t_2$. In other words, the time instance t_0 is chosen so that the interval $[t_0, t_2)$ is the longest interval, such that the processor is busy at all times during the interval and is assigned to a job instance with a deadline t_2 or earlier. From the argument in the preceding paragraph, it follows that the interval $[t_1, t_2)$ does satisfy the desired condition, but it may not be the longest such interval, and thus t_0 is chosen by extending this interval to the left as long as the schedule σ assigns time slots to job instances with deadlines t_2 or less.

By our choice of t_0, the processor is busy throughout the interval $[t_0, t_2)$, and thus it follows that the length of this interval equals the sum of the number of slots the schedule σ allocates to each job during this interval:

$$t_2 - t_0 \;=\; \sum_{K \in \mathcal{J}} \sigma(t_0, t_2, K) \qquad (2).$$

The next step in the proof aims to derive a bound on the value of each quantity $\sigma(t_0, t_2, K)$. For this purpose, we show that:

> **Claim:** If the schedule σ allocates a time slot in the interval $[t_0, t_2)$ to a job K, then the corresponding instance of K *lies entirely within the interval* $[t_0, t_2)$.

Consider an arbitrary job K and an instance j of this job. The interval corresponding to the instance K^j is $[\alpha(K, j), \alpha(K, j+1))$. If this interval is not contained within the interval $[t_0, t_2)$, then one of the following three cases can happen.

First, the interval $[\alpha(K, j), \alpha(K, j+1))$ has no overlap with the interval $[t_0, t_2)$. This can happen if $\alpha(K, j) \geq t_2$ or if $\alpha(K, j+1) \leq t_0$. In this case, the instance K^j is not relevant to scheduling during the interval $[t_0, t_2)$. In figure 8.5, these are the instances before the ath instance or after the cth instance of the job K.

A second possibility is that there is an overlap, but $\alpha(K, j+1) > t_2$ (see the cth instance of the job K in figure 8.5). We know that throughout the interval $[t_0, t_2)$, the schedule σ picks a job only if its current deadline does not exceed t_2. Throughout the interval $[\alpha(K, j), \alpha(K, j+1))$ corresponding to the instance K^j, the deadline is $\alpha(K, j+1)$, which exceeds t_2, and thus we can conclude that

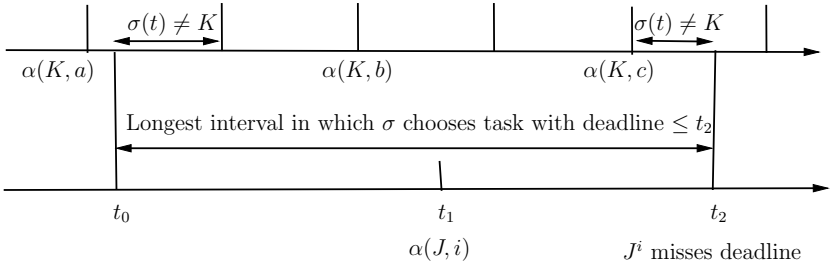

Figure 8.5: Proof of Test for EDF Schedulability

during the interval $[t_0, t_2)$, the schedule does not allocate any time slots to such an instance K^j.

The third and final possibility is that the interval corresponding to the instance K^j does overlap with $[t_0, t_2)$, the deadline $\alpha(K, j+1)$ does not exceed t_2, but $\alpha(K, j) < t_0$ (see the ath instance of the job K in figure 8.5). We claim that, in this case,

the job K is not ready at time t_0 in the schedule σ,

that is, all the demand for the instance K^j has been met during the interval $[\alpha(K, j), t_0)$, and the job K does not need anymore slots during the interval $[t_0, \alpha(K, j+1))$. Since the schedule picks only ready jobs, the claim implies that during the interval $[t_0, t_2)$, the schedule does not allocate slots to the instance K^j.

To prove the claim, assume to the contrary that the job K is ready at time t_0, that is, the instance K^j has not been allocated enough processing time before time t_0. Consider a time instance t such that $\alpha(K, j) \leq t < t_0$. The job K is ready at time t, and its deadline at time t is $\alpha(K, j+1)$. According to the EDF policy, then, $\sigma(t)$ cannot be $\perp$ and must be a job whose deadline at time t is $\leq \alpha(K, j+1)$. Since $\alpha(K, j+1) \leq t_2$, we can conclude that the interval $[\alpha(K, j), t_2)$ is an interval in which at every time the schedule picks a job with deadline t_2 or less. But this contradicts the assumption that $[t_0, t_2)$ is the longest such interval. In other words, if the job K is ready at time t_0, then we should be extending the interval of interest from $[t_0, t_2)$ to $[\alpha(K, j), t_2)$.

Thus, we have shown that whenever the EDF schedule σ picks a job K at a time instance t in the interval $[t_0, t_2)$, the corresponding instance of K must lie entirely within the interval $[t_0, t_2)$ (such as the bth instance of the job K in figure 8.5). The length of the interval $[t_0, t_2)$ is $t_2 - t_0$. The length of the interval corresponding to any instance of K is $\pi(K)$. It follows that the number of instances of K that lie entirely within $[t_0, t_2)$ is at most $(t_2 - t_0)/\pi(K)$. Note that the number of instances of K with a possible overlap with the interval $[t_0, t_2)$ can be the two extreme instances with partial overlap plus the number

of instances that lie entirely within it. But we have established that the two extreme instances are not allocated any slots.

To each instance of K that lies entirely inside the interval $[t_0, t_2)$, the schedule allocates at most $\eta(K)$ number of slots. It follows that, for every job K,

$$\sigma(t_0, t_2, K) \ \leq \ \eta(K) \cdot (t_2 - t_0)/\pi(K) \qquad (3).$$

The inequality (3) holds for the job J also. However, we know that the instance J^i is allocated strictly less than $\eta(J)$ number of slots by the schedule σ (since this instance misses its deadline t_2). This implies the following strict inequality:

$$\sigma(t_0, t_2, J) \ < \ \eta(J) \cdot (t_2 - t_0)/\pi(J) \qquad (4).$$

Summing the inequalities (3) over all the jobs and noting that the inequality is strict at least for the job J leads to:

$$\sum_{K \in \mathcal{J}} \sigma(t_0, t_2, K) \ < \ \sum_{K \in \mathcal{J}} \eta(K) \cdot (t_2 - t_0)/\pi(K) \qquad (5).$$

We can substitute U from equation (1) in the above inequality to get

$$\sum_{K \in \mathcal{J}} \sigma(t_0, t_2, K) \ < \ (t_2 - t_0) \cdot U \qquad (6).$$

From the equation (2) and inequality (6), we get

$$(t_2 - t_0) \ < \ (t_2 - t_0) \cdot U.$$

Since $t_2 - t_0$ is positive, we can conclude that $1 < U$, which is what we wanted to prove. ■

8.3 Fixed-Priority Scheduling

The EDF policy is an example of a *dynamic priority* policy, where different instances of the same job get assigned different priorities; as a result, at some time instance, a job J is preferred over another job K, while at some other time instance, the job K is preferred over the job J. Now we turn our attention to *fixed-priority* policies, where jobs are statically assigned fixed priorities, and whenever the scheduler has to make a choice, the job with the highest priority is chosen. In particular, we will analyze the properties of the most commonly used such policies, namely, the *deadline-monotonic* policy and the *rate-monotonic* policy.

8.3.1 Deadline-Monotonic and Rate-Monotonic Policies

Fixed-Priority Policies

A priority assignment for a set of jobs assigns each job a number such that no two jobs have the same number. Given two jobs J and K, if the number

assigned to the job J is larger than the number assigned to the job K, then the job J has a higher priority than the job K. Given such a priority assignment, a fixed-priority scheduling policy always prefers a job of higher priority over jobs with lower priorities.

More precisely, the fixed-priority schedule is constructed for each time $t = 0, 1, 2 \ldots$ in the following manner. Recall that, given a schedule up to time t, a job J is ready at time t if the instance of J that is active at time t has not already been allocated the necessary $\eta(J)$ number of slots. If there is no job that is ready at time t, then the next slot is left unassigned. Otherwise the fixed-priority schedule selects the job that is ready at time t and has the highest priority among all the jobs ready at that time.

The definition of such policies is summarized below.

FIXED-PRIORITY SCHEDULING POLICY

A *priority assignment* for a periodic job model $\mathcal{J}$ is a function ρ that maps each job $J \in \mathcal{J}$ to a natural number such that for every two distinct jobs J and K, $\rho(J) \neq \rho(K)$. A schedule σ for the periodic job model $\mathcal{J}$ is called a *fixed-priority schedule* with respect to the priority assignment ρ, if for every time $t \geq 0$, if no job is ready at time t in the schedule σ, then $\sigma(t) = \perp$, else $\sigma(t) = J$, such that the job J is ready at time t in the schedule σ, and for every job $K \in \mathcal{J}$, either the job K is not ready at time t in the schedule σ or $\rho(K) < \rho(J)$.

Given a priority assignment ρ, the corresponding fixed-priority schedule depends only on the ordering of the jobs induced by the priorities and not on the numerical values of priorities assigned to jobs. More precisely, consider two priority assignments ρ and ρ' for a periodic job model $\mathcal{J}$ such that for every pair of jobs J and K, $\rho(J) > \rho(K)$ exactly when $\rho'(J) > \rho'(K)$. Then at every time t, the fixed-priority scheduler makes exactly the same decision whether it is based on the priority assignment ρ or is based on the assignment ρ'. Thus, in this case, the fixed-priority schedule with respect to the assignment ρ coincides with the fixed-priority schedule with respect to the assignment ρ'.

Deadline-Monotonic and Rate-Monotonic Priorities

Once we commit to employing a fixed-priority scheduling policy, the only choice for a scheduling policy concerns the priority assignment to jobs. If the period of a job J is smaller than the period of another job K, then the instances of the job J arrive at a faster rate than the instances of the job K, and this suggests that the job J should be assigned a higher priority. This priority assignment rule is called *rate monotonic*: the priority assignment ρ for jobs is a rate-monotonic priority assignment if for every pair of jobs J and K, if $\pi(J) < \pi(K)$, then $\rho(J) > \rho(K)$. Note that if two jobs have the same period, then a rate-monotonic priority assignment still needs to assign different priorities to both, and there is a choice regarding which of the two should be assigned a higher priority.

When all deadlines are implicit, that is, for every job, its deadline equals its period, it turns out that the rate-monotonic policy is an optimal choice for assigning priorities. However, when the jobs have explicitly specified deadlines, then it seems intuitive to prefer a job with an earlier deadline over a job with a later deadline. The resulting priority assignment rule is called *deadline monotonic*: the priority assignment ρ for jobs is a deadline-monotonic priority assignment if for every pair of jobs J and K, if $\delta(J) < \delta(K)$, then $\rho(J) > \rho(K)$. Again, if two jobs have the same deadline, then a deadline-monotonic priority assignment can arbitrarily choose one of them to have a higher priority. In a case where the deadline of each job equals its period, the notions of deadline-monotonic policy and rate-monotonic policy coincide. Note that while the EDF policy also prefers job instances with earlier deadlines, EDF is a dynamic priority policy, whereas deadline monotonic is a fixed-priority policy.

The notions of rate-monotonic and deadline-monotonic schedules are formalized below.

Deadline-monotonic and Rate-monotonic Policy

A priority assignment ρ for a periodic job model $\mathcal{J}$ is called *deadline monotonic* if for all jobs $J, K \in \mathcal{J}$, if $\delta(J) < \delta(K)$, then $\rho(J) > \rho(K)$; and is called *rate monotonic* if for all jobs $J, K \in \mathcal{J}$, if $\pi(J) < \pi(K)$, then $\rho(J) > \rho(K)$. A schedule σ for the periodic job model $\mathcal{J}$ is called a *deadline-monotonic schedule* if there exists a deadline-monotonic priority assignment ρ, such that the schedule σ is a fixed-priority schedule with respect to ρ; and is called a *rate-monotonic schedule* if there exists a rate-monotonic priority assignment ρ, such that the schedule σ is a fixed-priority schedule with respect to ρ.

Examples

Let us revisit the job model consisting of the job J_1 with period 5, deadline 4, and WCET 3; and the job J_2 with period 3, deadline 3, and WCET 1. We know that this job model is schedulable: see figures 8.2 and 8.3 for deadline-compliant schedules. In particular, the schedule of figure 8.3 is an EDF schedule, and it sometimes prefers the job J_1 over J_2 and other times prefers the job J_2 over J_1. The priority assignment that assigns a higher priority to the job J_2 than to the job J_1 is deadline monotonic (and also rate monotonic). Observe that in the resulting fixed-priority schedule, the job J_2 is selected at time 0 as well as at time 3; as a result, the first instance of the job J_1 gets only two slots before its deadline 4, causing a missed deadline. If the priority assignment were to always prefer the job J_1 over the job J_2, then the first three slots are allocated to the job J_1, causing the first instance of the job J_2 to miss its deadline. Thus, fixed-priority scheduling policies do not produce deadline-compliant schedules for this job model.

Let us change the deadline of the job J_1 to 5: this leads to the job model with implicit deadlines consisting of the job J_1 with period 5 and WCET 3 and the

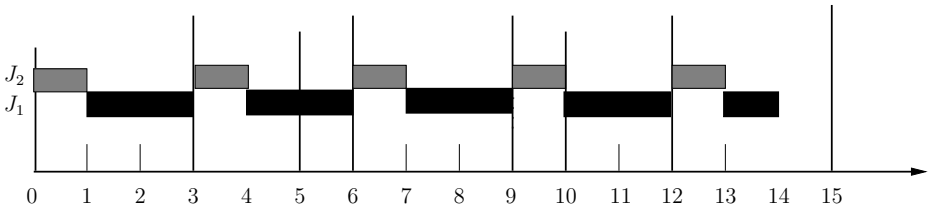

Figure 8.6: Illustrative Rate-Monotonic Schedule

job J_2 with period 3 and WCET 1. The rate-monotonic priority assignment still assigns a higher priority to the job J_2 than to the job J_1. The resulting fixed-priority schedule is shown in figure 8.6. The schedule is periodic with period 15 and meets all the deadlines.

Properties

We have already noted that a deadline-monotonic policy may fail to produce a deadline-compliant schedule even when the job model is schedulable. Conditions under which deadline-monotonic and rate-monotonic policies are guaranteed to succeed are studied in sections 8.3.2 and 8.3.3. For now let us note some basic properties of these policies.

The deadline-monotonic and rate-monotonic policies resolve the choice among the set of ready jobs by a simple local rule and, similar to the EDF policy, are examples of greedy algorithms. We also note that any fixed-priority scheduling policy is guaranteed to produce a periodic schedule with period equal to the least-common multiple of the periods of all the jobs. Furthermore, the schedules generated by these policies can be preemptive (for example, see the rate-monotonic schedule shown in figure 8.6).

The principal benefit of the deadline-monotonic as well as the rate-monotonic policy is that the overhead needed to implement the scheduler is minimal. The scheduler needs to assign a priority to each job off-line, and the computation necessary for this purpose involves ordering jobs according to their deadlines. As in the case of the EDF policy, the deadline-monotonic/rate-monotonic scheduling policy also needs to make a scheduling decision only when either the currently executing instance of a job finishes its execution or a new instance of another job arrives. The priority of an instance of a job equals the statically assigned priority to that job, and thus no computation is needed to determine priorities at run-time. As a result, the deadline-monotonic as well as the rate-monotonic policy can be easily incorporated in any operating system that supports priority-based scheduling of processes.

Exercise 8.9: Consider a periodic job model of exercise 8.6 consisting of the job J_1 with period 6, deadline 5, and WCET 2; the job J_2 with period 8, deadline

4, and WCET 2; and the job J_3 with period 12, deadline 8, and WCET 4. Is this model schedulable using the deadline-monotonic scheduling policy? ■

Exercise 8.10: Construct a periodic job model with two jobs such that the deadline-monotonic policy leads to a deadline-compliant schedule, but the rate-monotonic policy results in a schedule with missed deadlines. ■

8.3.2 Optimality of Deadline-Monotonic Policy *

We have already noted that, unlike the EDF policy, the deadline-monotonic policy is not an optimal policy for producing deadline-compliant schedules: a deadline-monotonic schedule for a schedulable periodic job model need not be deadline-compliant. In this section, we establish that the deadline-monotonic policy, however, is optimal among all fixed-priority policies: if there exists a priority assignment such that the corresponding fixed-priority schedule meets all the deadlines, then a deadline-monotonic schedule also meets all the deadlines. This implies that if we prefer fixed-priority scheduling for its simplicity and minimal scheduling overhead, then it is desirable to choose a deadline-monotonic priority assignment.

Criticality of First Instances

Before we prove optimality of the deadline-monotonic policy relative to fixed-priority schedules, let us establish a property that all fixed-priority schedules satisfy: if a fixed-priority schedule misses a deadline, then it misses the deadline of the first instance of some job. In other words, in a fixed-priority schedule, the first instances of all the jobs are the critical ones: if all first instances meet their deadlines, then the remaining instances are guaranteed to meet their deadlines. This implies that if we want to check whether a fixed-priority schedule is deadline-compliant, we can explicitly construct the schedule from time $t = 0$ up to time $t = d$, where d is the maximum of the deadlines of all the jobs, ensuring that no deadlines are missed until this time, instead of constructing and examining the entire periodic schedule up to time $t = p$, where p is the least-common multiple of the periods of all the jobs, which is typically much larger than d.

Theorem 8.4 [Criticality of the first instances in fixed-priority schedules] *If σ is a fixed-priority schedule for a periodic job model, and if the deadline of the first instance of every job is met in the schedule σ, then the schedule σ is deadline-compliant.*

Proof. Let $\mathcal{J}$ be a periodic job model, let ρ be a priority assignment for $\mathcal{J}$, and let σ be the fixed-priority schedule for $\mathcal{J}$ with respect to the assignment ρ. Let us assume that the schedule σ allocates enough slots to the first instance of every job. We want to prove that the schedule is deadline-compliant.

For proof by contradiction, assume that the schedule σ is not deadline-compliant. Then there must exist a job instance that misses its deadline in the schedule σ.

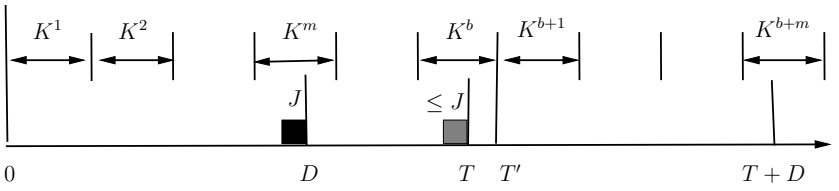

Figure 8.7: Illustration for Proof of Theorem 8.4

Let J be the job with the highest priority that misses a deadline. That is, there exists an instance of the job J that misses its deadline in the schedule σ, and if K is a job such that $\rho(K) > \rho(J)$, then all instances of the job K meet their deadlines in the schedule σ. Let h be the priority of this job J.

By assumption, the first instance of the job J gets $\eta(J)$ number of slots by its deadline. Let D be the time when the first instance finishes its execution, that is, D is the number such that $\sigma(0, D, J) = \eta(J)$ and $\sigma(D-1) = J$ (see figure 8.7 for illustration). By assumption, $D \leq \delta(J)$ holds.

For every $t \geq 0$, let $\theta(t)$ denote the number of slots that the schedule allocates to jobs with priority higher than that of the job J during the time interval of length D starting at time t:

$$\theta(t) \ = \sum_{K \in \mathcal{J},\, \rho(K) > h} \sigma(t, t + D, K).$$

For every instance a, the instance J^a of the job J arrives at time $t_a = \alpha(J, a)$. By definition, $\theta(t_a)$ is the number of slots that the schedule allocates to jobs of priority higher than h during the interval of length D starting at time t_a. Any slot not allocated to a job of higher priority is assigned to the job J as long as the job J needs slots, and thus if the condition $\theta(t_a) \leq D - \eta(J)$ holds, then the instance J^a finishes its execution within D units of its arrival and, thus, meets its deadline (since $D \leq \delta(J)$).

We know that the first instance of the job J finishes at time D, and thus $\theta(0) \leq D - \eta(J)$ holds. We also know that there exists an instance of the job J that misses its deadline, and thus for some a, $\theta(t_a) > D - \eta(J)$ must hold, which implies, $\theta(t_a) > \theta(0)$. Let T be the least time for which $\theta(T) > \theta(0)$; that is, if $\theta(t) > \theta(0)$, then $T \leq t$. Note that this particular time instance T cannot exceed the arrival time t_a of the a instance that misses its deadline, but T itself need not be the arrival time of any instance of the job J. By the choice of T, we know that $\theta(T-1) < \theta(T)$, and thus we can conclude that the schedule σ cannot choose a job of priority higher than h at time $T - 1$.

We proceed to show that the schedule σ does not allocate more slots to a job with priority higher than h during the interval $[T, T+D)$ than during the interval $[0, D)$:

Claim: if $\rho(K) > \rho(J)$ then $\sigma(T, T+D, K) \leq \sigma(0, D, K)$.

Consider a job K such that $\rho(K) > \rho(J)$. By assumption, all instances of such a job meet their deadlines. To prove the claim that $\sigma(T, T+D, K) \leq \sigma(0, D, K)$, our analysis depends on how many instances of the job K overlap with the interval $[0, D)$. Instances of the job K arrive every $\pi(K)$ time units. Let m be the integer such that $(m-1) \cdot \pi(K) < D \leq m \cdot \pi(D)$, that is, m is obtained by calculating $D/\pi(K)$, and if this quantity is a fraction, then rounding it up to the next integer. Then the first m instances of the job K overlap with the interval $[0, D)$ (see figure 8.7 for illustration).

Observe that the first $(m-1)$ instances of the job K lie entirely within the interval $[0, D)$, and the mth instance has a partial overlap. Note that since all instances of the job K meet their deadlines, the schedule σ allocates exactly $\eta(K)$ number of slots to each instance of the job K. Each of the first $(m-1)$ instances of the job K lie entirely within the interval $[0, D)$, and thus each of them contributes $\eta(K)$ to the quantity $\sigma(0, D, K)$. We know that the last slot of the interval $[0, D)$ is allocated to the job J. Since the job K has a higher priority than that of the job J, this implies that at time $(D-1)$, the job K is not ready, which implies that its corresponding instance has been allocated $\eta(K)$ number of slots by time $(D-1)$. This means that the contribution of the mth instance of K to the quantity $\sigma(0, D, K)$ is also $\eta(K)$. Thus, $\sigma(0, D, K)$ equals $m \cdot \eta(K)$.

Now let us focus our attention on how different instances of the job K overlap with the interval $[T, T+D)$. Suppose the earliest instance of the job K that arrives at time T or later is the $(b+1)$-th instance: $\alpha(K, b) < T \leq \alpha(K, b+1)$. Let T' be the arrival time of the $(b+1)$-th instance of the job K.

In general, $T' > T$ is possible, and then the bth instance of the job K also has an overlap with the interval $[T, T+D)$ (figure 8.7 illustrates this case). By the choice of T, we know that at time $(T-1)$, the schedule does not choose a job of priority higher than h and, thus, does not choose the job K or a job of priority higher than that of the job K. This can happen only if the job K is not ready at time $(T-1)$, which means that all the demand of the bth instance of the job K has been met by time $(T-1)$. This implies that even though the bth instance can overlap with the interval $[T, T+D)$, it does not contribute to the quantity $\sigma(T, T+D, K)$. This implies that $\sigma(T, T+D, K)$ equals $\sigma(T', T+D, K)$.

The number of instances of K that overlap with the interval $[T', T+D)$ is obtained by dividing its length by $\pi(K)$ and rounding up the answer to the next integer. Since $T' \geq T$, the length of the interval $[T', T+D)$ cannot exceed D, it follows that the number of instances of K that overlap with the interval $[T', T+D)$ is at most m (it is either m or $m-1$ depending on the difference between T and T'). Since an instance of the job K can be allocated only $\eta(K)$ number of slots, it follows that $\sigma(T', T+D, K) \leq m \cdot \eta(K)$.

We have established the claim that $\sigma(T, T+D, K) \leq \sigma(0, D, K)$ for every job K of priority higher than h. This means that $\theta(T) > \theta(0)$ is not possible, resulting in the desired contradiction. ■

Proof of Optimality

Now we proceed to prove that a deadline-monotonic policy is guaranteed to produce a deadline-compliant schedule as long as there exists a deadline-compliant fixed-priority schedule.

Theorem 8.5 [Optimality of deadline-monotonic policy] *Given a periodic job model $\mathcal{J}$, if there exists a priority assignment ρ for $\mathcal{J}$, such that the corresponding fixed-priority schedule σ is deadline-compliant, then every deadline-monotonic schedule for $\mathcal{J}$ is deadline-compliant.*

Proof. Let $\mathcal{J}$ be a periodic job model. Let ρ be a priority assignment for $\mathcal{J}$ such that the fixed-priority schedule σ with respect to ρ meets all the deadlines. Suppose the ordering of the jobs in $\mathcal{J}$ according to the priority assignment ρ is $J_1, J_2, \ldots J_n$, where n is the number of jobs in $\mathcal{J}$, with the job J_1 being the job of the highest priority and the job J_n being the job of the lowest priority.

Let ρ' be a deadline-monotonic priority assignment. We want to prove that the fixed-priority schedule with respect to the assignment ρ' is also deadline-compliant. If the ordering of the jobs according the priorities assigned by ρ' equals that according to the assignment ρ, then the schedule σ is also the fixed-priority schedule with respect to ρ' since the choices made by the schedule depend only on the relative priorities of the jobs and not on the numerical values of priorities. Hence, suppose that the ordering of the jobs in decreasing priorities according to the assignment ρ' is not the same as the ordering $J_1, J_2, \ldots J_n$. Then there must exist a pair of adjacent jobs J_a and J_{a+1} in this order such that their relative priorities are different according to the assignment ρ': $\rho'(J_{a+1}) > \rho'(J_a)$. Since the assignment ρ' is deadline-monotonic, it follows that the deadline of the job J_a is not earlier than that of the job J_{a+1}: $\delta(J_a) \geq \delta(J_{a+1})$.

First, we show that if we modify the priority assignment ρ by swapping the priorities of two such adjacent jobs, then the corresponding schedule is still deadline-compliant. That is, let ρ_1 be the priority assignment such that $\rho_1(J_b) = \rho(J_b)$ for $b < a$ and for $b > a+1$, $\rho_1(J_a) = \rho(J_{a+1})$ and $\rho_1(J_{a+1}) = \rho(J_a)$. Thus, the ordering of the jobs with decreasing priorities according to the assignment ρ_1 is $J_1, J_2, \ldots, J_{a-1}, J_{a+1}, J_a, J_{a+2}, \ldots J_n$.

Let σ_1 be the fixed-priority schedule with respect to the priority assignment ρ_1. We want to prove that the schedule σ_1 is deadline-compliant. From theorem 8.4, to prove that the schedule σ_1 meets the deadlines of all instances of all the jobs, it suffices to prove that it meets the deadline of the first instance of each job.

Consider a job J_b for $b = 1, 2, \ldots n$. Let D_b be the time by which the first instance of the job J_b finishes its execution in the deadline-compliant schedule σ according to the original priority assignment ρ. That is, $\sigma(D_b - 1) = J_b$ and $\sigma(0, D_b, J_b) = \eta(J_b)$. Since this is a deadline-compliant schedule, we know that $D_b \leq \delta(J_b)$ holds. We proceed to prove that, in the schedule σ_1 with respect to

the revised priority assignment ρ_1, for each $b \neq a$, the first instance of the job J_b finishes its execution by time D_b, and the first instance of the job J_a finishes its execution by time D_{a+1}. Since $D_{a+1} \leq \delta(J_{a+1})$ and $\delta(J_{a+1}) \leq \delta(J_a)$, it follows that the first instances of all the jobs finish their executions by their deadlines, implying deadline-compliance of the schedule σ_1.

To show that the first instance of the job J_a finishes its execution by time D_{a+1}, we prove:

> **Claim:** The schedule σ_1 allocates $\eta(J_a)$ time slots to the job J_a in the time interval $[0, D_{a+1})$.

Let us denote D_{a+1} by D. To prove the claim, let us consider a job J_b of a priority higher than that of the job J_a according to the priority assignment ρ_1 and compute how many time slots are allocated to the job J_b by the schedule σ_1 during the interval $[0, D)$. According to the priority assignment ρ_1, a job J_b has a higher priority than that of the job J_a if either $b < a$ or $b = a + 1$. We consider these two cases one by one.

- **Case $b < a$:** The job J_b has a higher priority than the job J_a in both the schedules σ and σ_1. Suppose the number of instances of the job J_b that overlap with the interval $[0, D)$ is m, that is, m is the integer obtained by rounding up the quantity $D/\pi(J_b)$. Using reasoning similar to the one used in the proof of theorem 8.4, we can conclude that $\sigma(0, D, J_b) = m \cdot \eta(J_b)$: each of the first $(m - 1)$ instances of the job J_b are entirely contained within the interval $[0, D)$, and since this schedule meets all the deadlines, each of them gets $\eta(J_b)$ number of slots; the mth instance can have a partial overlap, but since the priority of the job J_b is higher than that of the job J_{a+1} and the schedule σ chooses the job J_{a+1} at time $(D-1)$, this last instance of J_b must have been allocated all its demand before time D. Since m is the total number of instances of the job J_b that overlap with the interval $[0, D)$, the schedule σ_1 cannot possibly allocate more than $m \cdot \eta(J_b)$ number of slots to the job J_b, and we can conclude that $\sigma_1(0, D, J_b) \leq \sigma(0, D, J_b)$ holds.
- **Case $b = a + 1$:** Now the job J_b has a higher priority than the job J_a in the schedule σ_1 but has a lower priority than the job J_a in the schedule σ. We know that in the schedule σ the first instance of the job J_b finishes its execution by time D, and since this schedule is deadline-compliant, $\sigma(0, D, J_b) = \eta(J_b)$. It also implies that only the first instance of the job J_b has any overlap with the interval $[0, D)$, so it follows that in the schedule σ_1, even though the job J_b has a relatively higher priority, it cannot allocate more then $\eta(J_b)$ number of time slots to the job J_b. We can conclude that $\sigma_1(0, D, J_b) \leq \sigma(0, D, J_b)$ holds.

Thus, we have proved that for a job J_b that has a higher priority than the job J_a in the revised priority assignment ρ_1, the schedule σ_1 does not allocate more slots to the job J_b during the interval $[0, D)$ than the schedule σ does. Since

the schedule σ allocates $\eta(J_a)$ number of time slots to the job J_a by time D_a and $D_a \leq D$, it follows that $\sigma(0, D, J_a) = \eta(J_a)$. In the schedule σ_1, during the interval $[0, D)$, no job J_b for $b > a + 1$ will be chosen before the job J_a is allocated $\eta(J_a)$ number of time slots. Hence, the claim follows.

The proof that for each $b \neq a$, the first instance of the job J_b finishes by time D_b in the schedule σ_1 is similar and is left as an exercise.

We have established that the schedule corresponding to the priority assignment ρ_1 obtained by swapping priorities of an adjacent pair of jobs in the ordering given by the original priority assignment ρ is also deadline-compliant. If the ordering of the jobs according to the priority assignment ρ_1 equals the ordering according to the deadline-monotonic assignment ρ', we have already established our goal. If not, we can repeat the argument: in the assignment ρ_1, we can find a pair of jobs that are adjacent according to the ordering given by the assignment ρ_1 but have different relative ordering according to the assignment ρ' and swap their priorities to obtain the assignment ρ_2. By the argument above, deadline-compliance of the fixed-priority schedule σ_1 with respect to the assignment ρ_1 implies the deadline-compliance of the fixed-priority schedule σ_2 with respect to the assignment ρ_2.

To finish the proof, we need to establish that such swapping of jobs will not continue forever. To understand why only a bounded number of swaps suffice, let us consider a concrete example. Suppose the ordering of the jobs according to the original priority assignment ρ is J_1, J_2, J_3, J_4, and the ordering according to the desired (deadline-monotonic) priority assignment is J_3, J_1, J_4, J_2. Then starting from the assignment ρ, we can swap the ordering of the jobs J_2 and J_3 to get the priority assignment ρ_1 with the ordering J_1, J_3, J_2, J_4; then swap the ordering of the jobs J_1 and J_3 to get the priority assignment ρ_2 with the ordering J_3, J_1, J_2, J_4; and finally swap the ordering of the jobs J_2 and J_4 to get the priority assignment ρ'. As this example suggests, this process is the same as an algorithm for sorting a sequence of elements by swapping adjacent out-of-order elements, where each swap makes the current sequence more "similar" to the desired final sequence.

To make this argument precise, let us define the distance between two priority assignments ρ and ρ' to be the number of pairs (a, b), such that the relative ordering of the priorities of the jobs J_a and J_b are different in the two assignments. Such a distance can be at most $n(n-1)$, where n is the number of jobs. The assignment ρ_1 is obtained from the assignment ρ by swapping an adjacent pair of jobs to make the assignment more similar to the target assignment ρ'. More precisely, if the distance between the assignments ρ and ρ' is k, then the distance between the assignment ρ_1 and ρ' cannot be more than $(k-1)$: if the assignment ρ_1 is obtained from the assignment ρ by swapping the pair J_a and J_{a+1}, then the pairs that contribute to the distance between ρ_1 and ρ' are exactly those pairs that contribute to the distance between ρ and ρ', except the swapped pair $(a, a+1)$. Since the distance decreases by at least one at

every step, it follows that there can be at most $n(n-1)$ many swaps before the assignment becomes identical to ρ'. ■

Exercise 8.11: In the proof of theorem 8.5, the schedule σ_1 is the fixed-priority schedule obtained by swapping priorities of two adjacent jobs J_a and J_{a+1}, such that $\delta(J_a) \geq \delta(J_{a+1})$. We proved that in this schedule, the first instance of the job J_a finishes by time D_{a+1}, where for every b, D_b is the time by which the first instance of the job J_b finishes its execution in the original schedule σ. Complete the proof of deadline-compliance of the schedule σ_1 by showing that for every $b \neq a$, the first instance of the job J_b finishes its execution by time D_b in the schedule σ_1. ■

8.3.3 Schedulability Test for Rate-Monotonic Policy *

Given a periodic job model $\mathcal{J}$, how can we check if the rate-monotonic or the deadline-monotonic scheduling policy is going to succeed in producing a deadline-compliant schedule? One possibility is to explicitly compute the schedule and check if it meets all the deadlines. Theorem 8.4 assures us that it suffices to examine compliance of deadlines only for the first instances of all the jobs, and thus we need to compute the schedule only for the first d time slots, where d is the maximum of the deadlines of all the jobs. We proceed to establish a simpler condition based on the utilization that assures us that for a periodic job model with implicit deadlines, if the utilization is below a certain threshold value, then the rate-monotonic policy is guaranteed to produce a deadline-compliant schedule. Recall that the utilization of a job model specifies the fraction of available processing time that is needed to execute all the jobs and can be computed easily. We also know that the EDF policy is guaranteed to succeed as long as the utilization does not exceed 1. It turns out that the rate-monotonic policy is guaranteed to succeed as long as the utilization does not exceed 0.69. This result is of importance for both practical and theoretical reasons. In practice, if we know that the total demand for processing time is not too high (less than 69%, to be precise), then it suffices to employ the rate-monotonic policy, which has a minimal scheduling overhead. In theory, techniques used to establish this bound illustrate how to analyze algorithms for the worst case.

Analyzing Utilization for Two Jobs

For now, let us suppose that there are only two jobs with implicit deadlines. Let us call the job with the higher priority according to the rate-monotonic priority assignment J_1 and the job with the lower priority J_2. Let σ denote the corresponding fixed-priority rate-monotonic schedule. Let π_1 and π_2 be the periods of the two jobs. By assumption, $\pi_1 \leq \pi_2$. Let η_1 and η_2 be the worst-case execution times of the two jobs. The utilization for this job model is given by

$$U \;=\; \eta_1/\pi_1 \;+\; \eta_2/\pi_2.$$

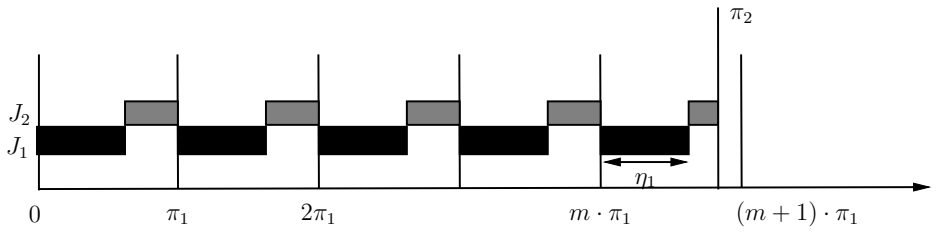

Figure 8.8: Analyzing Rate-monotonic Policy for Two Jobs: Case (a)

Our goal is to come up with a numerical bound B, which should be as high as possible, such that if $U \leq B$ holds, then we are guaranteed that the rate-monotonic schedule σ meets all the deadlines. For this purpose, we eliminate the four parameters in the above expression one by one, and the ensuing analysis reveals relationships among these parameters that cause the *worst-case* scenario for the rate-monotonic policy.

Eliminating WCET η_2

As the first step in the analysis, let us treat the parameters π_1, π_2, and η_1 as given and suppose we want to find out constraints on the fourth parameter η_2, in terms of these three fixed parameters so that the schedule is deadline-compliant.

For checking deadline-compliance, we need to examine only the first instances of the two jobs. Since the job J_1 has the higher priority, its first instance gets the first η_1 slots and is guaranteed to meet its deadline (recall that, by assumption, for every job, its WCET cannot exceed its deadline). To analyze the condition under which the job J_2 gets η_2 slots by its deadline π_2, we need to consider how many instances of the job J_1 are executed in the interval $[0, \pi_2)$. Let m be the number such that the condition $m \cdot \pi_1 \leq \pi_2 < (m+1) \cdot \pi_1$ holds, that is, m is obtained by rounding down the quantity π_2/π_1 to the nearest integer. Then the first m instances of the job J_1 lie entirely within the interval $[0, \pi_2)$.

To compute the number of slots available for the execution of the job J_2 before its deadline, there are two cases as shown in figures 8.8 and 8.9.

- **Case (a):** If $m \cdot \pi_1 + \eta_1 < \pi_2$, then the $(m+1)$th instance of the job J_1 finishes its execution before the deadline π_2 of the first instance of the job J_2. This is the case shown in figure 8.8. In this case, the total time allocated to the first job in the interval $[0, \pi_2)$ is $(m+1) \cdot \eta_1$. Then the first instance of the job J_2 can be allocated up to $\pi_2 - (m+1) \cdot \eta_1$ time slots. Thus, the schedule meets the deadlines as long as the condition $\eta_2 \leq \pi_2 - (m+1) \cdot \eta_1$ holds. This implies that the rate-monotonic policy succeeds as long as

$$U \ \leq \ \eta_1/\pi_1 + [\pi_2 - (m+1) \cdot \eta_1]/\pi_2.$$

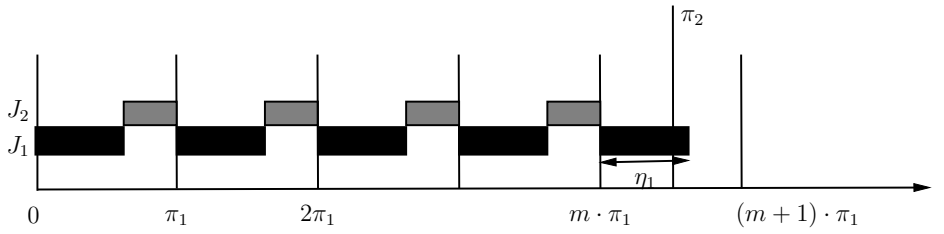

Figure 8.9: Analyzing Rate-monotonic Policy for Two Jobs: Case (b)

- **Case (b):** If $m \cdot \pi_1 + \eta_1 \geq \pi_2$, then the $(m+1)$th instance of the job J_1 does not finish its execution before the deadline π_2 of the first instance of the job J_2. In this case, the job J_2 meets its deadline only if it finishes execution before the $(m+1)$th instance of the job J_1 arrives (see figure 8.9), and the maximum number of slots that can be allocated to the job J_2 by its deadline is $m \cdot (\pi_1 - \eta_1)$. Thus, the schedule meets the deadlines as long as the condition $\eta_2 \leq m \cdot (\pi_1 - \eta_1)$ holds. This implies that the rate-monotonic policy succeeds as long as

$$U \ \leq \ \eta_1/\pi_1 + [m \cdot (\pi_1 - \eta_1)]/\pi_2.$$

Thus, in each case, we have obtained a bound B such that if the utilization is below that bound, the schedule is guaranteed to be deadline-compliant. This bound B can be viewed as a function of the three parameters, π_1, π_2, and η_1, and can be summarized as

$$\texttt{if}\ (m{\cdot}\pi_1{+}\eta_1 < \pi_2)\ \texttt{then}\ \eta_1/\pi_1{+}[\pi_2{-}(m{+}1){\cdot}\eta_1]/\pi_2\ \texttt{else}\ \eta_1/\pi_1{+}[m{\cdot}(\pi_1{-}\eta_1)]/\pi_2.$$

Eliminating WCET η_1

The next step is to eliminate the parameter η_1 by *minimizing* this function B over all possible choices of η_1: if the utilization is below this minimized value, then we are guaranteed that the utilization is below the desired bound no matter what value of η_1 is chosen, and hence the schedule is deadline-compliant.

For the case $m \cdot \pi_1 + \eta_1 < \pi_2$, we have

$$\begin{aligned} B &= \eta_1/\pi_1 + [\pi_2 - (m+1) \cdot \eta_1]/\pi_2, \\ &= 1 + \eta_1/\pi_1 - (m+1) \cdot \eta_1/\pi_2, \\ &= 1 - \eta_1 \cdot (m + 1 - \pi_2/\pi_1)/\pi_2. \end{aligned}$$

Since $\pi_2 < (m+1) \cdot \pi_1$, the quantity $(m + 1 - \pi_2/\pi_1)$ is positive. Hence, for given values of the parameters π_1 and π_2, the value of the bound B *decreases*

as η_1 increases. Hence, the minimum occurs when η_1 has the highest possible value, which due the condition $m \cdot \pi_1 + \eta_1 < \pi_2$, equals $\pi_2 - m \cdot \pi_1$.

For the case $m \cdot \pi_1 + \eta_1 \geq \pi_2$, we have

$$\begin{aligned} B &= \eta_1/\pi_1 + m \cdot (\pi_1 - \eta_1)/\pi_2, \\ &= m \cdot \pi_1/\pi_2 + \eta_1/\pi_1 - m \cdot \eta_1/\pi_2, \\ &= m \cdot \pi_1/\pi_2 + \eta_1 \cdot (\pi_2/\pi_1 - m)/\pi_2. \end{aligned}$$

Since $m \cdot \pi_1 \leq \pi_2$, the quantity $(\pi_2/\pi_1 - m)$ is positive. As a result, for given values of the parameters π_1 and π_2, the value of the bound B *increases* as η_1 increases. Hence, the minimum occurs when η_1 has the least possible value, which due the condition $m \cdot \pi_1 + \eta_1 \geq \pi_2$, equals $\pi_2 - m \cdot \pi_1$.

Thus, we have shown that the bound B is minimal when $\eta_1 = \pi_2 - m \cdot \pi_1$. Substituting this value of η_1 in the expression for B in the second case gives us the desired bound as a function of the parameters π_1 and π_2:

$$B \;=\; m \cdot \pi_1/\pi_2 + (\pi_2 - m \cdot \pi_1) \cdot (\pi_2/\pi_1 - m)/\pi_2.$$

We want to choose the parameters π_1 and π_2 to minimize this function. Note that m is the largest integer less than or equal to the ratio π_2/π_1. Let π_2/π_1 be $m + f$, where $f \in [0, 1)$ is the fractional part of this ratio. Substituting $m + f$ for the ratio π_2/π_1 in the above expression for the bound B leads to:

$$B \;=\; m/(m+f) \;+\; (1 - m/(m+f)) \cdot f \;=\; (m + f^2)/(m+f).$$

Computing the Numerical Bound

We have expressed the desired bound as a function of the two parameters m and f. Now we want to minimize this quantity over all choices of m and f, where m is a positive integer and f is a fraction in the interval $[0, 1)$.

$$B \;=\; (m + f + f^2 - f)/(m+f) \;=\; 1 + (f^2 - f)/(m+f).$$

Since $0 \leq f < 1$, the quantity $f^2 - f$ is always *negative*, and thus for a given value of the fraction f, the value of the bound B increases as m increases. Thus, the minium occurs at the smallest possible value of m. Note that, by assumption, $\pi_1 \leq \pi_2$, and thus m cannot be 0. This means that the smallest possible value of m is 1. Substituting this value in the expression for the bound B gives us

$$B \;=\; (1 + f^2)/(1+f).$$

This says that, for a given f, which is the fractional part of the ratio of the two periods, the schedule is deadline-compliant as long as the utilization does not exceed $(1 + f^2)/(1 + f)$.

The last step of the analysis is to minimize this function with respect to the parameter f, where $0 \leq f < 1$. For this purpose, let us *differentiate* the expression B with respect to f:

$$\begin{aligned} dB/df &= [2f(1+f) - (1+f^2)]/(1+f)^2, \\ &= (f^2 + 2f - 1)/(1+f)^2. \end{aligned}$$

Thus, the derivative dB/df is 0 exactly when $f^2 + 2f - 1 = 0$. This quadratic equation has two roots, $f = -1 - \sqrt{2}$ and $f = -1 + \sqrt{2}$, of which only $-1 + \sqrt{2}$ lies in the range $[0, 1)$ for f. Thus, the bound B is minimal when $f = -1 + \sqrt{2}$. Substituting this value in the expression B gives:

$$B \;=\; [1 + (-1 + \sqrt{2})^2]/(1 - 1 + \sqrt{2}) \;=\; 2(\sqrt{2} - 1).$$

This leads to the main claim for the case of two jobs:

> For a periodic job model with two jobs with implicit deadlines, a rate-monotonic schedule is guaranteed to be deadline-compliant if the utilization does not exceed $2(\sqrt{2} - 1)$.

Note that the quantity $2(\sqrt{2}-1)$ is 0.828. This means that with two jobs, if the utilization does not exceed 0.828, then we know that the rate-monotonic policy will succeed no matter what values of the periods and WCETs are chosen.

Understanding the Worst Case for Two Jobs

To recap the proof of the bound for two jobs, the worst case for the rate-monotonic policy, that is, the case where the utilization is as small as possible, while the resulting schedule is just barely deadline-compliant, occurs when (1) the period π_2 is $\sqrt{2}$ times the period π_1, (2) the WCET η_1 equals the difference $\pi_2 - \pi_1$, and (3) the WCET η_2 equals the difference $\pi_1 - \eta_1$.

Figure 8.10 shows this critical scenario for two jobs. For the job J_1, the period is 100, whereas for the job J_2, the period is 141. The WCET for the job J_1 is 41, whereas the WCET of the job J_2 is 59. Observe that the resulting schedule is deadline-compliant, and the utilization is $41/100 + 59/141$, which is about 0.828.

Suppose we increase the WCET of the first job to 42. The utilization for this updated job model is $42/100 + 59/141$, which is about 0.838, which exceeds the bound of the schedulability test. Observe that in the rate-monotonic schedule for this revised model, the second job gets only 57 slots by its deadline, and thus the schedule is not deadline-compliant.

Observe that the bound we have calculated is only a *sufficient* test for deadline-compliance. It may happen that the utilization for a model with two jobs exceeds the bound 0.828, and yet the rate-monotonic policy produces a deadline-compliant schedule. This can happen if the specific values of the periods and

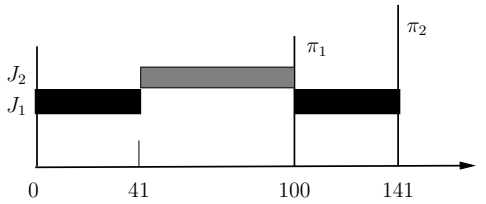

Figure 8.10: Worst Case for Rate-monotonic Policy with Two Jobs

WCETs in this model do not correspond to the worst-case scenario for the rate-monotonic policy. For example, for the job model consisting of the job J_1 with period 5 and WCET 3 and the job J_2 with period 3 and WCET 1, the utilization is 0.93, and yet as figure 8.6 illustrates, the rate-monotonic schedule meets all the deadlines.

Schedulability Test for n Jobs

The analysis for two jobs can be generalized to a job model with n jobs. We only state the worst-case scenario that the analysis reveals. Suppose the jobs are ordered $J_1, J_2, \ldots J_n$ in a decreasing order of priorities (that is, in an increasing order of periods) according to the rate-monotonic policy. For each job J_a, let us denote its period (and deadline) by π_a and its WCET by η_a. Then the worst case for the rate-monotonic policy occurs when the following relationships hold among the different parameters:

$$\begin{aligned}
\pi_1 &< \pi_2 < \cdots < \pi_n < 2 \cdot \pi_1, \\
\eta_1 &= \pi_2 - \pi_1, \\
\eta_2 &= \pi_3 - \pi_2, \\
&\cdots \\
\eta_n &= \pi_1 - (\eta_1 + \eta_2 + \cdots + \eta_{n-1}) = 2 \cdot \pi_1 - \pi_n, \\
\pi_2/\pi_1 &= \pi_3/\pi_2 = \cdots = \pi_n/\pi_{n-1} = 2^{1/n}.
\end{aligned}$$

If we calculate the utilization for these values of the parameters, then we get the bound $B_n = n(2^{1/n} - 1)$.

This scenario for the case of $n = 3$ is shown in figure 8.11. The WCETs and periods are chosen so that the first instances of all three jobs finish by the time the second instance of the first job arrives, the second instance of the first job finishes by the time the second instance of the second job arrives, which finishes by the time the second instance of the third job arrives, which finishes by the time the third instance of the first job arrives. The relationship among the three periods π_1, π_2, and π_3 and the three WCETs η_1, η_2, and η_3 is given by:

$$\pi_2 = 1.26 \cdot \pi_1,$$

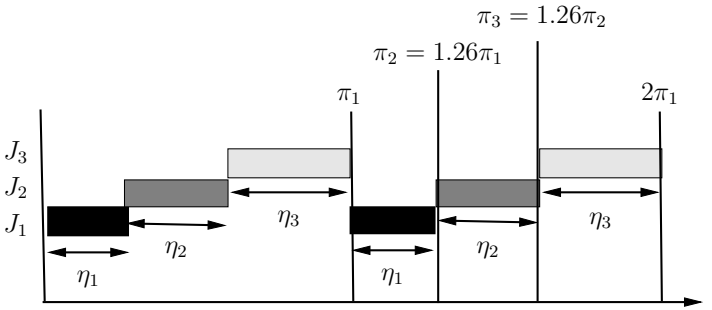

Figure 8.11: Worst Case for Rate-monotonic Policy with Three Jobs

$$\begin{aligned}
\pi_3 &= 1.26 \cdot \pi_2 = 1.59 \cdot \pi_1, \\
\eta_1 &= \pi_2 - \pi_1 = 0.26 \cdot \pi_1, \\
\eta_2 &= \pi_3 - \pi_2 = 0.33 \cdot \pi_1, \\
\eta_3 &= 2 \cdot \pi_1 - \pi_3 = \pi_1 - (\eta_1 + \eta_2) = 0.41 \cdot \pi_1, \\
B_3 &= \eta_1/\pi_1 + \eta_2/\pi_2 + \eta_3/\pi_3 = 0.78.
\end{aligned}$$

The following theorem summarizes the bound for n jobs:

Theorem 8.6 [Schedulability test for Rate-Monotonic Policy] *Given a periodic job model with n jobs with implicit deadlines, if the utilization does not exceed the quantity* $B_n = n(2^{1/n} - 1)$*, then every rate-monotonic schedule is guaranteed to be deadline-compliant.* ■

Note that the bound B_n decreases as n increases. The table in figure 8.12 shows the values of these bounds for $n = 1, 2, \ldots 10$. This means, for instance, when we have six jobs, if the utilization is 0.735 or less, then we are guaranteed that the rate-monotonic policy produces a deadline-compliant schedule.

Finally, let us consider the *limit* of the expression $n(2^{1/n} - 1)$. This limit turns out to be *ln* 2, the natural logarithm of 2, and equals 0.69. That is, for every number n, the value of the expression $n(2^{1/n} - 1)$ is at least 0.69. This means:

> If the utilization of a periodic job model with implicit deadlines is 0.69 or less, then a rate-monotonic schedule is deadline-compliant.

Exercise 8.12: Consider a periodic job model with three jobs with implicit deadlines: the job J_1 with period 4 and WCET 1, the job J_2 with period 6 and WCET 2, and the job J_3 with period 8 and WCET 3. Can we conclude that the rate-monotonic policy results in a deadline-compliant schedule using the utilization-based schedulability test? Does the rate-monotonic policy result in a deadline-compliant schedule? ■

n	1	2	3	4	5	6	7	8	9	10
B_n	1	0.828	0.780	0.757	0.743	0.735	0.729	0.724	0.721	0.718

Figure 8.12: Utilization Bound for Rate-monotonic Policy

Exercise 8.13*: Let $\mathcal{J}$ be a periodic job model, let ρ be a priority assignment for $\mathcal{J}$, and let σ be the fixed-priority schedule for $\mathcal{J}$ with respect to the assignment ρ. Prove that the schedule σ is deadline-compliant if the following condition is satisfied for every job J:

$$\delta(J) \;\geq\; \eta(J) \;+\; \sum_{K \in \mathcal{J}:\rho(K)>\rho(J)} \lceil (\delta(J)/\pi(K)) \rceil \cdot \eta(K)$$

In this formula, for a rational number f, $\lceil f \rceil$ denotes the integer obtained by rounding up f, that is, the smallest integer that is greater than or equal to f. ■

Bibliographic Notes

Scheduling algorithms for real-time systems is a well-studied topic: see [SAÅ+04] for a survey. The key result developed in this chapter, namely, the optimality and the analysis of the rate-monotonic scheduling policy, is due to Liu and Layland [LL73]. The presentation in this chapter is based on [But97] (see also [Liu00]). We also refer the reader to [FMPY06, BDL+11] for formalization of job models and schedulability analysis using the computational model of timed processes and reachability analysis for timed automata.

We have only briefly discussed the problem of estimating the worst-case execution time of tasks, which is also a well-studied problem with multiple theoretical approaches and tools (see [WEE+08] for a survey).

Real-time scheduling is supported by a number of operating systems (see [RS94] and [Kop00]).