Fundamentals of Real-Time Systems

Jian-Jia Chen

May 20, 2019

Contents

Preface

1 Introduction

2

Terminology and Basic Scheduling Theory

This chapter provides an overview of the algorithmic, task, and system models that will be used in the book. Task models will be given in Section 2.2. The concept of different schedules is formally defined in Section 2.3 to cover uniprocessor and multiprocessor schedulers, which will be widely used in the book to conduct many proofs. Section 2.4 provides an overview of different classifications and characteristics of scheduling algorithms. We will extend the well-known triplet notation of scheduling-theoretical problems to real-time systems in Section 2.5. Section 2.6 provides the overview regarding sufficient, necessary, and exact schedulability tests. Metrics that are adopted for theoretical comparisons of scheduling algorithms and schedulability tests are introduced in Section 2.7. The chapter is concluded with the scheduling/test anomaly and the needs of sustainable timing analyses in Section 2.8

2.1 Mathematical Notation

This section provides an overview of the notation used in this book.

- $\mathbb{R}$ is the set of real numbers, and $\mathbb{N}$ is the set of natural numbers.
- $[m]$ denotes the set of positive integers ranging from 1 to m, i..e, $[m] = \{1, 2, \ldots, m\}$.
- $\mathbb{1}_{\text{condition}}$ is a binary indicator. If the condition holds, the value is 1; otherwise the value is 0. This is widely used in this book to identify whether a job is executed in a schedule at time t.
- All the variables regarding tasks, jobs, platforms are assumed non-negative. Deadlines, and periods are assumed positive real numbers.

- Although we can informally state that a job is executed from time t to time $t + \Delta$, this has to be formally defined to avoid confusion. In this book, an interval of execution is denoted by a mixed (time) interval that is open at the beginning and closed at the end, i.e., $(t, t+\Delta]$. This is very important when we define the schedules and certain properties. This will be further clarified in Section 2.3.

2.2 Aperiodic, Sporadic, and Periodic Tasks

The fundamentals of real-time systems start from the model of computation and the system under consideration. Essentially, we consider a set of programs that can be concurrently executed. Towards this, we have to first classify several basic terms. An *algorithm* is the logical procedure to solve a certain problem. It informally specifies a sequence of elementary steps that an execution machine must follow to solve the problem, but it is not necessarily (and usually not) expressed in a formal programming language. A *program* is the implementation of an algorithm in a programming language and can be executed several times with different inputs. A *process* or *job* is an instance of a program that produces a set of outputs given a sequence of inputs. A process is launched and managed usually by the operating system. An operating system (OS) is a program that acts as an intermediary between a user of a computer and the computer hardware by providing interfaces, provides an "abstraction" of the physical machine (for example, a file, a virtual page in memory, etc.), manages the access to the physical resources of a computing machine, makes the computer system convenient to use, executes user programs and makes solving user problems easier, etc.

In this book, a job is the basic entity for scheduling. The timing parameters of a job J_j include:

- Arrival time a_j or release time r_j: time the job becomes ready for execution
- Computation time C_j (or execution time): time necessary to the processor for executing the job without interruption
- Absolute deadline d_j: time at which the job should be completed
- Relative deadline $D_j = d_j - r_j$: time between the arrival time and the absolute deadline
- Start time s_j: time the job starts its execution

- Finishing time f_j: time the job finishes its execution
- Response time $R_j = f_j - r_j$: time length at which the job finishes its execution after its arrival

When jobs (usually with the same computation requirement) are released recurrently, e.g., a control task as illustrated in Figure 2.1, these jobs can be modeled by a recurrent task. A widely-used deterministic recurrent task model is the so-called periodic real-time task model, also called the Liu and Layland task model presented by Liu and Layland [40] in 1973. In such a model, a periodic task τ_i releases an infinite number of *task instances* (*jobs*) periodically under a given *period* T_i, where the first job of task τ_i is released at its *phase* (or offset) O_i. Due to the recurrent releases, all the jobs of task τ_i should have an abstraction of the execution time. This is typically characterized by its *worst-case execution time* (WCET) C_i without any interruption. The basic assumption is that the jobs of a task are from a finite set of executable program instances. Therefore, it is possible to derive an upper bound on their execution times to have a succinct timing model. How to derive the worst-case execution time of a program safely will be discussed in Chapter 6. Each periodic task also associates with a *relative deadline* D_i. When a job of task τ_i arrives at time t, its *absolute deadline* is $t + D_i$.

In contrast, the sporadic real-time task model defined by Mok [42] is more flexible, in which any two consecutive releases of jobs of task τ_i are temporally separated by at least the *minimum inter-arrival time* T_i of task τ_i. Under the sporadic real-time task model, it is unnecessary to specify the phase of task τ_i, as the release pattern is not deterministic.

A periodic/sporadic task set **T** is called with 1) *implicit deadlines*, if $D_i = T_i$ holds for any τ_i in **T**, 2) *constrained deadlines*, if $D_i \leq T_i$ holds for any τ_i in **T**, and 3) *arbitrary deadlines*, otherwise. Since it is possible that $D_i > T_i$, it is possible that there are more than one job of task τ_i in the ready queue. In general, we assume that the jobs of a sporadic/periodic task are executed in the first-come-first-serve (FCFS) manner. Otherwise, a job released earlier by task τ_i may starve if the subsequent jobs of task τ_i can be executed before it. This assumption is very natural in uniprocessor systems.

The utilization $U_i = C_i/T_i$ of task τ_i defines the percentage of time a processor has to execute task τ_i when a sufficient number of jobs of task τ_i arrive periodically. In general, for hard real-time systems, we will implicitly assume that $C_i \leq \max\{D_i, T_i\}$; otherwise, there is no chance to meet the deadline of task τ_i in the worst case. For a periodic/sporadic task set **T**, we can define the *hyper-period* of **T**, which is the least common multiple (LCM)

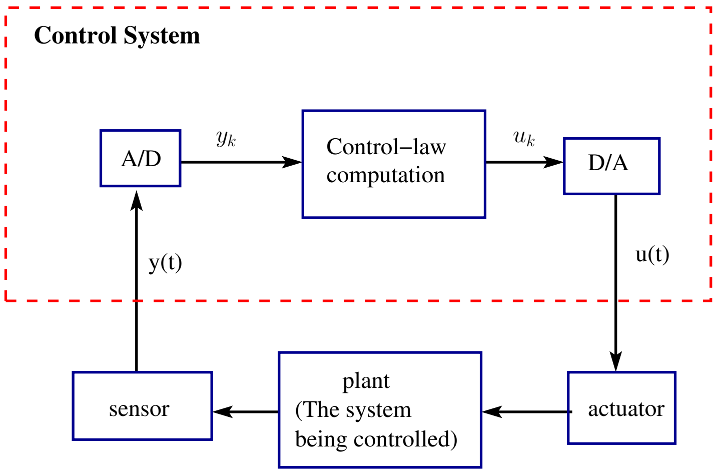


Figure 2.1: A periodic/sporadic control task

of the minimum inter-arrival times (periods) of the tasks in **T**. In this book, we will also implicitly assume that the hyper-period of **T** exists. For periodic task systems, the hyper-period can be used to repeat the same job arrival pattern every hyper-period. It is not obvious why we also need this definition now for sporadic task systems, since the argument of the repetitive pattern does not hold. But, we will touch this topic later in Chapter 4.

We now take a closer look into the implementation of such recurrent tasks. In List 2.1, the body of the control task in Figure 2.1 is implemented as a loop. To realize the periodic release of two jobs with a period T, the programmer intends to calculate the remaining amount of time “timeToSleep” until the next release. This may seem to create a periodic task, but the resulting release pattern is however not periodic but sporadic. The issue here is that this recurrent task may be *preempted* by other higher-priority jobs/tasks (or interrupted by hardware interrupts) between the two statements “end := get the system tick;” and “sleep timeToSleep”. Such occurence would lead to wrong sleep amount of time being called. However, if timeToSleep is always non-negative, the resulting release pattern would be sporadic with the minimum inter-arrival time T.

```
while (true) {
  start := get the system tick;
  perform analog_to_digital conversion to get y;
  compute control output u;
  output u and do digital_to_analog conversion;
  end := get the system tick;
  timeToSleep := T−(end−start);
  sleep timeToSleep;
}
```

Listing 2.1: Sporadic Control System

To resolve the above problem due to preemptions and interrupts, one possibility to create a periodic task is to use a system call sleepUntil(), which suspends the task until the next release, as shown in List 2.2. Such a system call typically exists in real-time operating systems with similar names, e.g., vTaskDelayUntil() in FreeRTOS. This creates a periodic pattern as long as the time when sleepUntil() is called is always no more than the corresponding nextRelease. Otherwise, the job may be released too late. The key issue here is the *overrun handling* in real-time operating systems. If the current time $t >$ nextRelease, some RTOSes may discard the next released job as this is considered to be too late, whilst some RTOSes may still release the next job even though it is already late.

For periodic real-time tasks, dropping the release of a job would result in incorrect semantics, especially when we consider arbitrary-deadline task systems, i.e., $D_i > T_i$ for some task τ_i. For uniprocessor systems, it is not problematic if a job of task τ_i is not released before all its previous jobs of task τ_i finish their executions since we assume their executions are based on the FCFS policy.

```
while (true) {
  perform analog_to_digital conversion to get y;
  compute control output u;
  output u and do digital_to_analog conversion;
  nextRelease := phase + period T;
  sleepUntil nextRelease;
}
```

Listing 2.2: Periodic Control System: sleepUntil

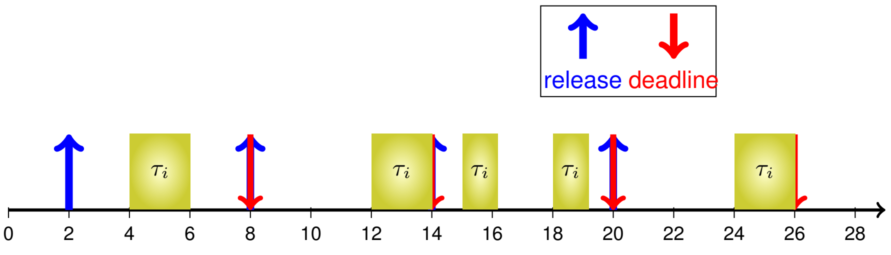
Another possibility to implement a periodic task is to set a periodic timer to interrupt the operating system periodically, as illustrated in List 2.3. However, such an implementation requires additional effort to maintain the potential multiple activations of the same program at the same time and to ensure the FCFS policy for the jobs of a periodic task when $D_i > T_i$.

```
set timer to interrupt periodically with period T;
at each timer interrupt {
  perform analog_to_digital conversion to get y;
  compute control output u;
  output u and do digital_to_analog conversion;
}
```

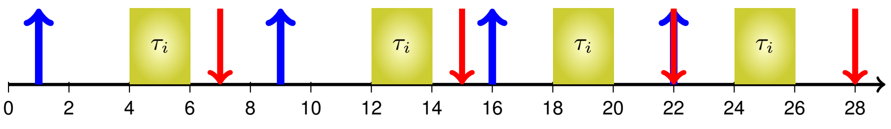
Listing 2.3: Periodic Control System: Periodic Timer

We should also keep in mind that it is in general impossible to allocate a unique hardware timer for a periodic task as modern computers only have limited hardware timers. In most RTOSes, usually only one hardware timer is allocated to handle the job releases, and all the periodic/sporadic tasks should share the hardware timer. Implementations like List 2.2 can be found in FreeRTOS, and implementations like List 2.3 can be found in RTEMS. Discussions about overrun handling if $D_i > T_i$ for a certain task τ_i in RTEMS and FreeRTOS can be found in [16].

Figure 2.2 provides examples of the above two recurrent task models. We will discuss more fine-grained recurrent task models in Chapter 9.



(a) Periodic task $\tau_i = \{C_i = 2, T_i = D_i = 6, O_i = 2\}$



(b) Sporadic task $\tau_i = \{C_i = 2, T_i = D_i = 6\}$

Figure 2.2: Examples of periodic and sporadic task models

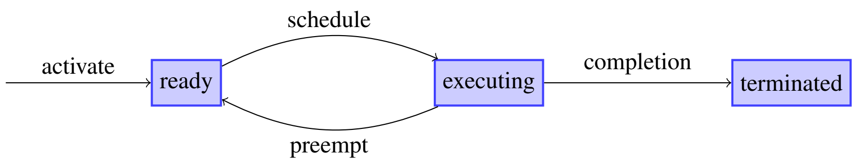


Figure 2.3: Job States

2.3 Schedules

Since the execution entities (tasks, processes, threads, etc.) are competing with each other, scheduling policies are needed to decide when to schedule an entity, which entity to schedule, and how to schedule entities. So, what is a scheduling algorithm (scheduler)? To answer this question, we will first discuss the schedules and scheduling algorithms with respect to jobs in uniprocessor systems and multiprocessor systems in Sections 2.3.1 and 2.3.2, respectively. Then, we extend the concept to periodic and sporadic real-time task systems in Section 2.3.3. Moreover, in Sections 2.3.1, 2.3.2, and 2.3.3, we focus on hard real-time guarantees to define the feasibility of schedules and the schedulability of the set of jobs or tasks. We then extend the definition to worst-case response time analyses in Section 2.3.4 and soft real-time systems in Section 2.3.5.

2.3.1 Scheduler: Uniprocessor Job-Level Perspectives

For uniprocessor systems, we assume that at most one job is executed at a time. Therefore, a **scheduling algorithm** (or **scheduler**) determines the order that jobs execute on the processor, call a schedule. Particularly, jobs (in a simplified version) may be in one of three states listed in Figure 2.3.

A schedule is an assignment of the given jobs to the processor, such that each job is executed (not necessarily consecutively) until completion. Suppose that $\mathbf{J} = \{J_1,\ J_2,\ \dots\ J_n\}$ is a set of n given jobs. A schedule for $\mathbf{J}$ can be defined as a function $\sigma : \mathbb{R} \rightarrow \mathbf{J} \cup \{\perp\}$, where $\sigma(t) = J_j$ denotes that job J_j is executed at time t, and $\sigma(t) = \perp$ denotes that the system is idle at time t.

If $\sigma(t)$ changes its value at some time t, the processor performs a **context switch** at time t. For a schedule σ to be valid with respect to the arrival time, the absolute deadline, and the execution time of the given jobs, we need to have the following conditions for each J_j in $\mathbf{J}$ for *hard real-time guarantees*:

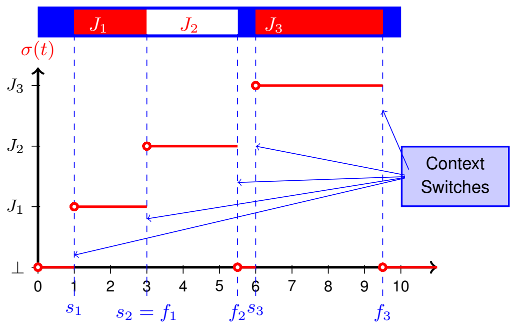
- $\sigma(t) \neq J_j$ for any $t \leq r_j$ and $t > d_j$ and,
- $\int_{r_j}^{d_j} \mathbb{1}_{\sigma(t)=J_j} dt = C_j$, where $\mathbb{1}_{\sigma(t)=J_j}$ evaluates to 1 if $\sigma(t) = J_j$ and 0 otherwise.

Note that the integration $\int$ of $\mathbb{1}_{\sigma(t)=\textit{certain job}}$ over time used in this book is only a symbolic representation for summation. If the above conditions are satisfied, we say that the schedule is *feasible* for the set **J** of jobs with respect to specified timing constraints for hard real-time guarantees. A set **J** of jobs is *schedulable* if there exists a feasible schedule for the set of jobs. A scheduling algorithm is *optimal* for hard real-time guarantees if it always produces a feasible schedule when one exists (under any scheduling algorithm).

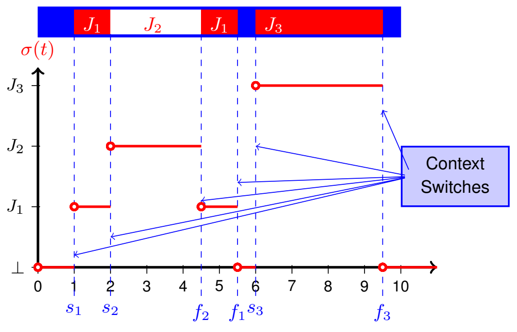
We sometimes informally state that a job is executed from time t to time $t + \Delta$. To keep the notion consistent, in this book, an interval of execution is denoted by a mixed (time) interval that is open at the beginning and closed at the end, i.e., $(t, t + \Delta]$. For example, in the schedule in Figure 2.4, job J_1 is executed in time interval $(1, 3]$, job J_2 is executed in time interval $(3, 5.5]$, and job J_3 is executed in time interval $(6, 9.5]$.

Some other constraints may also be introduced. In such cases, the schedule should also respect to those constraints. For example, we can consider the non-preemptive and preemptive schedules. A schedule is **non-preemptive** if a job cannot be preempted by any other jobs, i.e., only one interval with $\sigma(t) = J_j$ for every job J_j in **J**. A schedule is **preemptive** if a job can be preempted, i.e., more than one interval with $\sigma(t) = J_j$ for any job J_j in **J** are allowed. Figure 2.4 illustrates a preemptive and a non-preemptive schedule.

As we have already introduced the concept of worst-case execution times, it is also possible that we do not know the exact execution time of a job but its WCET. Therefore, depending on the actual execution times of the jobs, a scheduling algorithm may produce different schedules. For example, consider three jobs $J_1 = \{r_1 = 0, d_1 = 10, C_1 = 3\}$, $J_2 = \{r_2 = 3, d_2 = 5, C_2 = 2\}$, and $J_3 = \{r_3 = 2, d_3 = 8, C_3 = 3\}$ scheduled by a simple algorithm which schedules the job with the lowest index in the ready queue in a non-preemptive manner whenever the processor idles. Figure 2.5 presents two resulting schedules of the above scheduling algorithm. In Figure 2.5a, J_1 finishes at time 2. Since J_3 is the only job in the ready queue at time 2, J_3 is executed non-preemptively until it finishes at time 5. Although job J_2 arrives at time 3 and has a higher priority than J_3, job J_2 has to wait until time 5 to start its execution since the schedule is non-preemptive. In Figure 2.5b, J_1 finishes at time

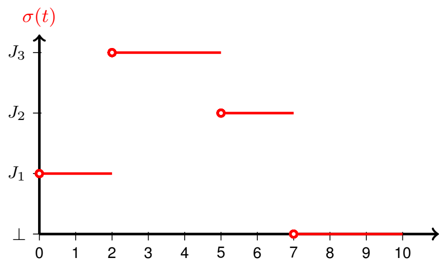


(a) A Non-Preemptive Schedule

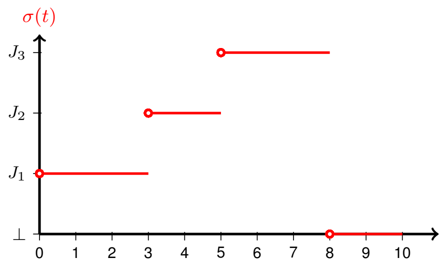


(b) A Preemptive Schedule

Figure 2.4: Preemptive and non-preemptive schedules



(a) Non-Preemptive: J_1 runs for 2 time units



(b) Non-Preemptive: J_1 runs for 3 time units

Figure 2.5: Non-preemptive schedules: J_1 has a higher priority than J_2 and J_2 has a higher priority than J_3, where $r_1 = 0, d_1 = 10, C_1 = 3$, $r_2 = 3, d_2 = 5, C_2 = 2$, and $r_3 = 2, d_3 = 8, C_3 = 3$. The scheduler executes the highest-priority job in the ready queue.

3. Since J_2 and J_3 are both in the ready queue at time 3, the higher-priority job J_2 is executed non-preemptively until it finishes at time 5.

In the above example, job J_2 misses its deadline in the schedule in Figure 2.5a and the schedule in Figure 2.5b is feasible. As demonstrated above, a scheduling algorithm may produce a feasible schedule when all the jobs are executed in their worst-case execution times. Therefore, we would have to alter the definition of feasibility and schedulability of a scheduling algorithm a bit to handle such cases.

▶ **Definition 2.1.** Suppose that we are given a set **J** of jobs, in which C_j is the worst-case execution time, r_j is the arrival time, d_j is the absolute deadline of job J_j in **J**. A schedule σ is feasible for hard real-time guarantees when the actual execution time of J_j is C'_j (i.e., $0 \leq C'_j \leq C_j$) for each J_j in **J** if

- $\sigma(t) \neq J_j$ for any $t \leq r_j$ and $t > d_j$ and,
- $\int_{r_j}^{d_j} \mathbb{1}_{\sigma(t)=J_j} dt = C'_j$.

The set **J** of jobs is *schedulable* for hard real-time guarantees under a scheduling algorithm if the resulting schedule is always feasible for any combinations of the actual execution time $0 \leq C'_j \leq C_j$ of job J_j in **J**. A scheduling algorithm is *optimal* for hard real-time guarantees if it always produces feasible schedule(s) when the set **J** is schedulable under a scheduling algorithm. ◀

2.3.2 Scheduler: Multiprocessor Job-Level Perspectives

In multiprocessor platforms, a schedule also involves an assignment of the given jobs to the processors, such that each job is executed until completion. In this section, we assume that we are given m processors. For the simplicity of presentation, these m processors are supposed to be *identical* (homogeneous) in this section. Suppose that $\mathbf{J} = \{J_1, J_2, \dots J_n\}$ is a set of n given jobs. A schedule for **J** can be defined as a function $\sigma : \mathbb{R} \times [m] \rightarrow \mathbf{J} \cup \{\perp\}$, where $\sigma(t, \ell) = J_j$ denotes that job J_j is executed on processor ℓ at time t, and $\sigma(t, \ell) = \perp$ denotes that processor ℓ is idle at time t.

In the book, we do not allow parallel execution of a job. That is, a job cannot be executed on two processors at the same time. (This constraint will be relaxed when we visit the DAG task model in Chapter 15.) Since we assume that the execution of a job must be *sequential*, we know that at most one processor $\ell \in [m]$ can have $\sigma(t, \ell) = J_j$ at any time point t.

For a *partitioned* multiprocessor schedule, a job has to be executed only on one processor. That is, if $\sigma(t, \ell) = J_j$ for a certain t and ℓ, then $\sigma(t', \ell') \neq J_j$ for any t' and $\ell' \neq \ell$. For a *global* multiprocessor schedule, a job can be executed on different processors. Therefore, it is possible that $\sigma(t, \ell) = J_j$ and $\sigma(t', \ell') = J_j$ for some $t \neq t'$ and $\ell \neq \ell'$.

Now, we can extend the definition of schedulability and optimality of scheduling algorithms in Definition 2.1 for multiprocessor systems.

▶ **Definition 2.2.** We are given m identical processors and a set **J** of jobs, in which C_j is the worst-case execution time, r_j is the arrival time, d_j is the absolute deadline of job J_j in **J**. A schedule σ is feasible for hard real-time guarantees when the actual execution time of J_j is C'_j (i.e., $0 \leq C'_j \leq C_j$) for each J_j in **J** if

- at most one processor $\ell \in [m]$ has $\sigma(t, \ell) = J_j$ at any $t \in \mathbb{R}$,
- $\sigma(t, \ell) \neq J_j$ for any $t \leq r_j$, $t > d_j$, and $\ell \in [m]$, and
- $\sum_{\ell \in [m]} \int_{r_j}^{d_j} \mathbb{1}_{\sigma(t,\ell)=J_j} dt = C'_j$.

The set **J** of jobs is *schedulable* for hard real-time guarantees under a scheduling algorithm if the resulting schedule is always feasible for any combinations of the actual execution time $0 \leq C'_j \leq C_j$ of job J_j in **J**. A scheduling algorithm is *optimal* for hard real-time guarantees if it always produces feasible schedule(s) when the set **J** is schedulable under a scheduling algorithm. ◀

2.3.3 Scheduler: Sporadic/Periodic Tasks' Perspectives

The definitions of schedulers and scheduling algorithms in the previous subsections are based on the jobs' perspective. We now extend these definitions to sporadic and periodic task systems. For a given sporadic task set **T**, each task τ_i in **T** can generate an infinite number of jobs as long as the temporal conditions of arrival times of the jobs generated by task τ_i can satisfy the minimum inter-arrival time constraint.

Suppose that the j^{th} job generated by task τ_i is denoted as $J_{i,j}$. Let the set of jobs generated by task τ_i be denoted as $\mathbf{FJ}_i$. A feasible set of jobs generated by a sporadic real-time task τ_i should satisfy the following conditions:

- By the definition of the WCET of task τ_i, the actual execution time $C_{i,j}$ of job $J_{i,j}$ is no more than C_i, i.e., $C_{i,j} \leq C_i$.
- By the definition of the relative deadline of task τ_i, we have $d_{i,j} = r_{i,j} + D_i$ for any integer j with $j \geq 1$.
- By the minimum inter-arrival time constraint, we have $r_{i,j} \geq r_{i,j-1} + T_i$ for any integer j with $j \geq 2$.

A feasible set of jobs generated by a periodic real-time task τ_i should satisfy the first two conditions above and the following condition:

- By periodic releases, we have $r_{i,1} = O_i$ and $r_{i,j} = r_{i,j-1} + T_i$ for any integer j with $j \geq 2$.

A *feasible collection* $\textbf{\textit{FJ}}$ *of jobs* generated by a task set $\mathbf{T}$ is the union of the feasible sets of jobs generated by the sporadic (or periodic) tasks in $\mathbf{T}$, i.e., $\mathbf{FJ} = \cup_{\tau_i \in \mathbf{T}} \mathbf{FJ}_i$. It should be obvious that there are infinite feasible collections of jobs generated by a sporadic real-time task set $\mathbf{T}$.

For a feasible collection $\mathbf{FJ}$ of jobs generated by $\mathbf{T}$, a uniprocessor schedule for $\mathbf{FJ}$ can be defined as a function $\sigma : \mathbb{R} \rightarrow \mathbf{FJ} \cup \{\perp\}$, where $\sigma(t) = J_{i,j}$ denotes that job $J_{i,j}$ is executed at time t, and $\sigma(t) = \perp$ denotes that the system is idle at time t. Recall that we assume that the jobs of task τ_i should be executed in the FCFS manner. Therefore, if $\sigma(t) = J_{i,j}$ then $\sigma(t') \notin \{J_{i,h} | h = 1, 2, \ldots, j-1\}$, for any $t' > t$ and $j \geq 2$.

Similar to Definition 2.1, the feasibility and optimality of scheduling algorithms should be defined based on all possible feasible collections of jobs generated by $\mathbf{T}$.

▶ **Definition 2.3.** Suppose that we are given a set $\mathbf{T}$ of sporadic real-time tasks on a uniprocessor system. A schedule σ of a feasible collection $\mathbf{FJ}$ of jobs generated by $\mathbf{T}$ is feasible for hard real-time guarantees if the following conditions hold for each $J_{i,j}$ in $\mathbf{FJ}$:

- $\sigma(t) \neq J_{i,j}$ for any $t \leq r_{i,j}$ and $t > d_{i,j}$,
- $\int_{r_{i,j}}^{d_{i,j}} \mathbb{1}_{\sigma(t)=J_{i,j}} dt = C_{i,j}$, and
- if $\sigma(t) = J_{i,j}$ then $\sigma(t') \notin \{J_{i,h} | h = 1, 2, \ldots, j-1\}$, for any $t' > t$ and $j \geq 2$.

A sporadic real-time task set $\mathbf{T}$ is *schedulable* for hard real-time guarantees under a scheduling algorithm if the resulting schedule of any feasible collection $\mathbf{FJ}$ of jobs generated by $\mathbf{T}$ is always feasible. A scheduling algorithm is *optimal* for hard real-time guarantees if it always produces feasible schedule(s) when the task set $\mathbf{T}$ is schedulable under a scheduling algorithm. ◀

The definition regarding schedulability and optimality for periodic task systems is the same. The definition is not presented explicitly.

We can also extend the concept to multiprocessor systems. Again, here, we assume that we have m *identical* (homogeneous) processors. For a feasible collection $\mathbf{FJ}$ of jobs generated by $\mathbf{T}$, a schedule for $\mathbf{FJ}$ can be defined as a function $\sigma : \mathbb{R} \times [m] \rightarrow \mathbf{FJ} \cup \{\perp\}$, where $\sigma(t, \ell) = J_{i,j}$ denotes that job $J_{i,j}$ is executed on processor ℓ at time t, and $\sigma(t, \ell) = \perp$ denotes that the processor ℓ is idle at time t. Recall that we assume that the jobs of task τ_i should be executed in the FCFS manner. Therefore, if $\sigma(t, \ell) = J_{i,j}$ then

$\sigma(t', *) \notin \{J_{i,h} | h = 1, 2, \ldots, j-1\}$, for any $t' > t$ and $j \geq 2$, where $*$ implies any of the m processors.

For a *partitioned* multiprocessor schedule, all jobs generated by a task have to be executed only on one processor. That is, if $\sigma(t, \ell) = J_{i,j}$ for a certain t and ℓ, then $\sigma(t', \ell') \neq J_{i,h}$ for any t', integer $\ell' \neq \ell$, and integer h. For a *clustered* multiprocessor schedule, all jobs generated by a task have to be executed only on a subset of processors. For a *global* multiprocessor schedule, a job can be executed on different processors.

2.3.4 Scheduler: Worst-Case Response Time Analyses

In some cases, the relative deadlines of the sporadic/periodic tasks do not have to be specified. Or, alternatively, even with a deadline miss, the designer is interested to know the worst-case response time (WCRT) of a task, which is the upper bound on the response times of the jobs generated by a sporadic/periodic task under a given scheduling algorithm. In this case, the feasibility and optimality of scheduling algorithms cannot be defined as described above since the relative deadlines can be absent. We therefore need to define the worst-case response time (WCRT) here as well.

▶ **Definition 2.4.** Suppose that we are given a set **T** of sporadic real-time tasks on a uniprocessor system. A schedule σ of a feasible collection **FJ** of jobs generated by **T** has a worst-case response time (WCRT) R_i of a task τ_i if the following conditions hold for each $J_{i,j}$ in **FJ**:

- $\sigma(t) \neq J_{i,j}$ for any $t \leq r_{i,j}$ and $t > r_{i,j} + R_i$,
- $\int_{r_{i,j}}^{r_{i,j}+R_i} \mathbb{1}_{\sigma(t)=J_{i,j}} dt = C_{i,j}$, and
- if $\sigma(t) = J_{i,j}$ then $\sigma(t') \notin \{J_{i,h} | h = 1, 2, \ldots, j-1\}$, for any $t' > t$ and $j \geq 2$.

The WCRT of a task τ_i in **T** under a scheduling algorithm is R_i if the above condition holds in the resulting schedule of any feasible collection **FJ** of jobs generated by **T**. ◀

The above definition is for sporadic real-time tasks on uniprocessor systems, which can be easily extended to periodic tasks and multiprocessor systems.

2.3.5 Scheduler: Soft Real-Time Perspectives

For soft real-time systems, some occasional deadline misses are possible and acceptable. In this book, we do not explicitly discuss soft real-time systems, but it would be meaningful to discuss the difference shortly. One issue is how to handle a job with a deadline miss. Should the job be aborted or should the job be continued? If the job is aborted, the actual execution time of the job may be shorter than its actual execution requirement. However, this may result in an inconsistent system state. The subsequent computation or control may be incorrect. If the job is continued, other subsequent jobs may also miss their deadlines. This may result in overloading of the system. In both cases, a schedule remains feasible if $\int_{r_{i,j}}^{r_{i,j}+D_i} \mathbb{1}_{\sigma(t)=J_{i,j}} dt < C_{i,j}$. For the former case, $\sigma(t) \neq J_{i,j}$ for any $t \leq r_{i,j}$ and $t > r_{i,j} + D_i$. For the latter case, it is possible that $\sigma(t) = J_{i,j}$ for some $t > r_{i,j} + D_i$.

This is not going to be further discussed in the book.

2.4 Classification of Scheduling Algorithms

In this section, we briefly review different classes of scheduling algorithms. The classification here is not meant to be complete. For classical scheduling theory, please refer to the book by Pinedo [47].

2.4.1 Preemptive vs. Non-preemptive

The scheduling algorithm can use preemptive or non-preemptive scheduling policies. A schedule σ is **non-preemptive** if a job cannot be preempted by any other jobs, i.e., only one interval with $\sigma(t) = J_j$ for every job J_j in **J**. A schedule σ is **preemptive** if a job can be preempted, i.e., more than one interval with $\sigma(t) = J_j$ for any job J_j in **J** are allowed (but not enforced). The difference has been illustrated in Figure 2.4. Note that a preemptive scheduling algorithm may produce a non-preemptive schedule, depending on the job arrivals.

When preemptions are allowed, the scheduler can allocate the processor to jobs that need to finish as early as possible to meet their deadlines. However, under non-preemptive scheduling, such jobs may experience long blocking times and miss their deadlines due to non-preemptive scheduling. Therefore, (optimal) preemptive scheduling algorithms usually have better performance than (optimal) non-preemptive scheduling algorithms. Moreover, sometimes a problem can be easily handled with preemptions, whilst non-preemptive execution can be very difficult to deal with. We will demonstrate such a gap of optimality in Chapter 3. However, to date, there is no guarantee that a problem with preemptions is always easier than the corresponding problem without preemptions, see for example Figure 2.7 by Pinedo [47].

Although preemptive scheduling algorithms may be preferred at the first glance. In practice, preemptions create additional context switch overhead and may destroy the worst-case execution time analysis. The correct calculation of the WCET of a task is not easy if preemption is allowed, as preemption introduces additional overhead to the system, including suspending the task, inserting it into the ready queue, flushing the processor pipeline, and dispatching the new incoming task. Non-preemptive scheduling may also be enforced by the hardware. For example, messages in control area network (CAN) buses are not preemptable.

Therefore, both of them are widely used in academia and industry, depending on the execution platforms and the tolerance of the context switch overheads. In this book, both of them will be detailed.

2.4.2 Work-Conserving

The scheduling algorithm can use work-conserving or non-work-conserving scheduling policies. A schedule σ is **work-conserving** if one of the jobs in the ready queue is always executed. In uniprocessor systems, $\sigma(t) \neq \perp$ if there is at least a job in the ready queue at time t. In multiprocessor systems under global scheduling, $\sigma(t, \ell) \neq \perp$ if there is a job in the ready queue at time t but not executed at time t (also called list scheduling in multiprocessor systems). In contrast, a schedule σ is **non-work-conserving** if the schedule does not execute any job even if there is already one ready job in the ready queue.

The intuition behind work-conserving scheduling algorithms is quite obvious. If there is any work to be done and the processor is idle, we should just perform some work instead of letting the processor idle. Work-conserving scheduling algorithms are implemented in modern real-time operating systems. However, this does not mean that work-conserving scheduling algorithms are always the best. In some scenarios, work-conserving scheduling algorithms in fact are not the best strategies. Let us recall the example used in Figure 2.5. The schedule in Figure 2.5a is work-conserving, and job J_2 misses its deadline. If we simply let the processor idle from 2 to 3 even when J_1 finishes at time 2, the resulting schedule is feasible for all the three jobs. We will discuss this in more detail in Section 3.3.

2.4.3 Fixed-priority (Static-priority) vs. Dynamic-priority

Under dynamic-priority scheduling, the priority of a task may change over time. Under fixed-priority (FP) scheduling, each task is assigned a unique priority before execution and does not change over time. The jobs generated by a task always have the same priority defined by the task under FP scheduling. Under dynamic-priority or fixed-priority scheduling, the schedulability policy is very simple. Whenever there are jobs in the ready queue, the highest-priority job in the ready queue is executed. Such an execution of a job can be preemptive or non-preemptive.

Well-known dynamic-priority scheduling algorithms are earliest-deadline-first (EDF) and least-laxity-first (LLF) scheduling algorithms (to be analyzed in Chapter 3). In EDF, the job whose absolute deadline is the earliest has the highest priority. Therefore, it is possible that a job of task τ_1 has a higher priority than a job of task τ_2 and another job of task τ_1 has a lower priority than the same job of task τ_2.

Well-known fixed-priority assignments are rate-monotonic (RM) priority ordering and deadline-monotonic (DM) priority ordering (to be analyzed in Chapter 4). For example, under RM priority ordering, priorities are assigned to tasks according to their request rates. That is, tasks with higher request rates (i.e., shorter periods) have higher priorities, in which ties are broken arbitrarily.

For FP scheduling, in this book, we assume that there are sufficient priority levels offered by the operating system so that each task has a unique priority. When task τ_i has a higher priority than task τ_j, we denote their priority relationship as $\tau_i > \tau_j$. Since the priority levels are unique, the priority assignment is a total order. We will use the following three task sets:

- $hp(\tau_k)$ is the set of higher-priority tasks than task τ_k,
- $hep(\tau_k)$ is $hp(\tau_k) \cup \{\tau_k\}$, and
- $lp(\tau_k)$ is the set of lower-priority tasks than task τ_k.

It can be easily shown that preemptive fixed-priority (FP-P) scheduling is in fact not optimal with respect to schedulability of real-time tasks. Consider the following periodic real-time tasks: $\tau_1 = \{C_1 = 2, T_1 = D_1 = 4, O_1 = 0\}$ and $\tau_2 = \{C_2 = 5, T_2 = D_2 = 10, O_2 = 0\}$. Since there are only two periodic tasks, we have only two different priority orderings.

- **Case 1**: $\tau_1 > \tau_2$, in this case, the first job of τ_2 misses its deadline since only 4 units of its execution time is executed in time intervals $(2, 4]$ and $(6, 8]$.
- **Case 2**: $\tau_2 > \tau_1$, in this case, the first job of τ_1 misses its deadline.

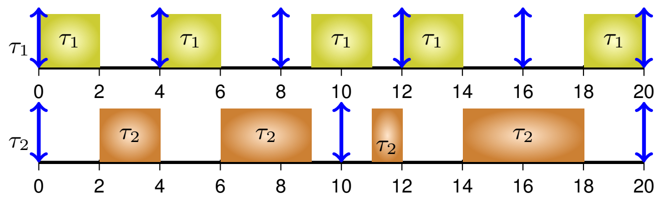


Figure 2.6: FP-Scheduling is not optimal

Therefore, no matter which priority assignment is used, one of the two tasks misses its deadline. However, the EDF schedule (demonstrated in Figure 2.6) shows that the task set is in fact schedulable under a dynamic-priority scheduling policy, which assigns a higher priority to the first job of τ_2 than the third job of τ_1.

2.4.4 Offline vs. Online

Scheduling algorithms can also be classified into offline (static) and online (dynamic) scheduling algorithms. For an **offline** (static) scheduling algorithm, the scheduling decisions take a priori knowledge about arrival times, execution times, and deadlines into account. The dispatcher allocates the processor when it is interrupted by the timer. The schedule is stored in a table, and the timers controlled by the table are generated at design time. For an **online** (dynamic) scheduling algorithm, the scheduling decisions are based on the jobs that have arrived to the system and do not have any knowledge about the jobs that will be released in the future. An online algorithm assumes that the knowledge of a job is revealed once it arrives. (Here, we probably still do not know the actual execution time of a job, but we can assume that a safe worst-case execution time is revealed when a job arrives.)

For offline scheduling algorithms, the context switches and scheduling decisions are usually **time-triggered**. According to Kopetz [32], "*In an entirely time-triggered system, the temporal control structure of all tasks is established a priori by off-line support-tools. This temporal control structure is encoded in a Task-Descriptor List (TDL) that contains the cyclic schedule for all activities of the node. This schedule considers the required precedence and mutual exclusion relationships among the tasks such that an explicit coordination of the tasks by the operating system at run time is not necessary. . . . The dispatcher is activated by the synchronized clock tick. It looks at the TDL, and then performs the action that has been planned for this instant.*"

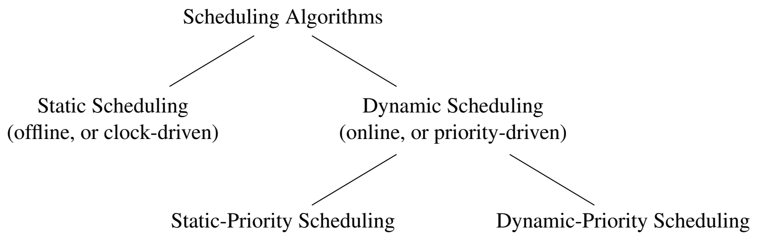

Scheduling Algorithms
Static Scheduling
(offline, or clock-driven)
Dynamic Scheduling
(online, or priority-driven)
Static-Priority Scheduling
Dynamic-Priority Scheduling

Figure 2.7: Classification of Scheduling Algorithms

The disadvantage of offline scheduling algorithms is that the response to sporadic or aperiodic events may be poor.

Online scheduling is the only option if the future workload is unpredictable. For sporadic real-time task systems, the actual inter-arrival time of two jobs of task τ_i is in fact unknown. Therefore, offline scheduling is only applicable with high response time. Since online scheduling is based on the jobs/events that arrive to the system, such scheduling algorithms are usually **event-triggered**. Both dynamic-priority and fixed-priority scheduling strategies are online scheduling.

In scheduling theory, a **clairvoyant** algorithm is a (hypothetical) algorithm that knows the future and makes the best possible decisions regardless of the required time/space complexity. Although such a clairvoyant algorithm may not always exist, it can be used for comparing the performance of real algorithms against the best possible one.

2.5 Definition of Scheduling Problems

Over the last sixty years, there has been a considerable amount of research effort on scheduling aperiodic tasks. In the domain of scheduling theory, a scheduling problem is described by a triplet $\text{Field}_1|\text{Field}_2|\text{Field}_3$.

- Field_1: describes the machine environment.
- Field_2: specifies the processing characteristics and constraints.
- Field_3: presents the objective to be optimized.

Since the machine environment is unique, the Field_1 field contains only one entry. The Field_2 field can have multiple entries if there are multiple constraints or characteristics. The Field_3 field often has a single entry.

2.5.1 Notation of Scheduling Theoretical Problems

Since the real-time system community and the scheduling theory community use different terminologies, it would be meaningful to provide a unified notation in this book to avoid unnecessary confusion.

Field$_1$

In this book, we consider the following possible machine environments:

- 1: indicates a single processor
- P_m: indicates m **identical** (also called *homogeneous*) parallel processors, in which all m processors have the same characteristics. That is, the execution time of a job is independent from the processor it is executed on.
- Q_m: indicates m **related** (also called *uniform*) processors with different but related performances. The performance of a processor is defined by a scaling factor of the speed. That is, the execution time of a job J_j on a processor ℓ is C_j/s_ℓ, where s_ℓ is the speed scaling factor of processor ℓ. The amount of work done in time interval $(r_j, d_j]$ in a schedule $\sigma()$ is $\sum_{\ell\in[m]}\int_{r_j}^{d_j} s_\ell \mathbb{1}_{\sigma(t,\ell)=J_j} dt$.
- R_m: indicates m **unrelated** (also called *heterogeneous*) processors with different and unrelated performance. The execution time of a job J_j on a processor ℓ is defined as $C_{j,\ell}$. There is no assumption regarding the relative performance. A processor can be faster to run a certain job and slower for another job. Such a setting is important to model processors with different instruction set architectures (ISA).
- F_m: indicates m processors in series, called **flow shop**. Each job has to be processed on processor 1, then on processor 2, etc., on processor m. Flow shop scheduling is one of the most fundamental scheduling theory.

Field$_2$

In this book, we consider the following execution constraints and characteristics:

- $prmp$: The scheduling policy is preemptive scheduling.
- r_j: The aperiodic jobs have specified arrival times (as well as absolute deadlines in our context).

- d_j: The aperiodic jobs have specified absolute deadlines.
- fp: The scheduling policy is fixed-priority scheduling.
- $period$: The task set is periodic real-time task systems (potentially with different offsets if $O_i = 0$ is not specified).
- $O_i = 0$: The tasks in a set have the same offset.
- $spor$: The task set is sporadic real-time task systems.
- $harmonic$: Harmonic task systems [34] represent a special case of sporadic/periodic real-time task systems. In a harmonic task system, for any two tasks τ_i and τ_j, the minimum inter-arrival time (or period) T_i is an integer multiple of T_j if $T_i > T_j$. As presented by Kramer et al. [33], the periods of the control applications in automotive systems can be modified to be harmonic with minor changes.
- $impl$: The task set is an implicit-deadline task set.
- $cons$: The task set is a constrained-deadline task set.
- arb: The task set is an arbitrary-deadline task set.
- $prec$: The jobs/tasks have precedence constraints.
- $mutex$: The tasks/jobs are not independent. Shared resources are guarded by using mutual exclusion locks (mutex) or binary semaphores.
- $global$: The scheduling policy is multiprocessor global scheduling.
- $partitioned$: The scheduling policy is multiprocessor partitioned scheduling.
- $cluster$: The scheduling policy is multiprocessor clustered scheduling.

Note that we use the following default characteristics in this book:

- It is assumed to be aperiodic jobs if $spor$ or $period$ is not specified.
- It is assumed to be non-preemptive if $prmp$ is not specified.
- It is assumed to allow dynamic-priority scheduling if fp is not specified.

- Periodic tasks are assumed to have different offsets if $period$ is present and $O_i = 0$ is not specified.
- Tasks/jobs are assumed independent from each other unless $mutex$ or $prec$ is specified.

Field$_3$
In this book, we consider the following objectives:

- $C_{\max}$: The objective is to minimize the completion time of the jobs, called **makespan** in scheduling theory. This is not compatible with periodic/sporadic task systems in general. But, in multiprocessor scheduling, we will translate the task partition problem for implicit-deadline sporadic real-time tasks to a similar setting to the well-known makespan problem.
- $L_{\max}$: The objective is to minimize the maximum lateness among the given jobs. The lateness of a job J_j is its finishing time f_j minus its absolute deadline d_j, i.e., $f_j - d_j$. The goal is to minimize $\max_{J_j} f_j - d_j$.
- $\leq D$: The objective is to meet the deadlines of the jobs/tasks.
- RT: The objective is to analyze the safe worst-case response time (WCRT) of the jobs/tasks for the given Field$_1$ and Field$_2$ setting.

2.5.2 Examples of Scheduling Theoretical Notation

The following examples illustrate the notation:

▶ **Example 2.5.** $1|prmp|L_{\max}$: This problem deals with a uniprocessor system. The input is a set of jobs released at time 0 with different absolute deadlines. The schedule can be preemptive. The objective is to minimize the maximum lateness. ◀

▶ **Example 2.6.** $1|r_j|L_{\max}$: This problem deals with a uniprocessor system. The input is a set of jobs with different release times and different absolute deadlines. The schedule is non-preemptive. The objective is to minimize the maximum lateness. ◀

▶ **Example 2.7.** $1|period, impl, prmp|{\leq}D$: This problem deals with a uniprocessor system. The input is a set of implicit-deadline periodic real-time tasks with different offsets. The schedule can be preemptive. The objective is to meet the deadlines of all the jobs generated by the periodic tasks. ◀

▶ **Example 2.8.** $1|spor, impl, prmp, fp|_{\leq}D$: This problem deals with a uniprocessor system. The input is a set of implicit-deadline sporadic real-time tasks. The schedule is preemptive fixed-priority (FP-P). The objective is to meet the deadlines of all the jobs generated by the sporadic real-time tasks. ◀

▶ **Example 2.9.** $P_m||C_{\max}$: This problem deals with an identical (homogeneous) multiprocessor system. The input has m identical processors and a set of jobs released at time 0. The objective is to find a partitioned multiprocessor schedule so that the makespan is minimized. ◀

▶ **Example 2.10.** $P_m|prec|C_{\max}$: This problem deals with an identical (homogeneous) multiprocessor system. The input has m identical processors and a set of jobs released at time 0 under precedence constraints. The objective is to find a partitioned multiprocessor schedule so that the makespan is minimized. ◀

▶ **Example 2.11.** $P_m|spor, impl, prmp, partitioned|_{\leq}D$: This problem deals with an identical (homogeneous) multiprocessor system. The input has m identical processors and a set of implicit-deadline sporadic real-time tasks. The objective is to find a partitioned multiprocessor schedule so that all the jobs generated by the sporadic real-time tasks can meet their deadlines. ◀

2.6 Schedulability Tests

Two separate but co-related problems are studied in real-time systems: 1) how to *design scheduling policies* to feasibly schedule the tasks, referred to as the *scheduler design* problem as mentioned above already, and 2) how to *validate* the schedulability of a scheduling algorithm, referred to as the *schedulability test* problem.

A *schedulability test* of a scheduling algorithm validates whether a given task system is schedulable by the scheduling algorithm. A schedulability test is referred to as *sufficient* if all of the task systems that it deems schedulable are in fact schedulable. Similarly, a schedulability test is referred to as *necessary* if all of the task systems that it deems unschedulable are in fact unschedulable. Schedulability tests that are both sufficient and necessary are referred to as *exact*. Exact schedulability tests provide the most precise classification of task systems; however, they are typically more difficult to derive and may have much higher computational complexity than generally simpler sufficient tests or necessary tests. We can rephrase the above definitions as follows:

- **Sufficient schedulability test** $\mathcal{C}$: if condition $\mathcal{C}$ holds for the input task set $\mathbf{T}$ on the platform, then $\mathbf{T}$ is schedulable under the schedulability algorithm on the platform.

- **Necessary schedulability test** $\mathcal{C}'$: if $\mathbf{T}$ is schedulable under the schedulability algorithm on the platform, then condition $\mathcal{C}$ holds for the input task set $\mathbf{T}$ on the platform.

- **Exact schedulability test** $\mathcal{C}^*$: the input task set $\mathbf{T}$ is schedulable under the schedulability algorithm on the platform if and only if condition $\mathcal{C}^*$ holds for the input task set $\mathbf{T}$ on the platform.

Therefore, for a fixed scheduling algorithm and a fixed platform, we can conclude that 1) a task set that can pass a necessary schedulability test $\mathcal{C}'$ also passes an exact schedulability test $\mathcal{C}^*$, and 2) a task set that can pass an exact schedulability test $\mathcal{C}^*$ also passes a sufficient schedulability test $\mathcal{C}$.

2.7 Theoretical Comparisons of Scheduling Algorithms and Schedulability Tests

The performance of schedulability tests and scheduling algorithms for real-time systems can be compared in a number of different ways. These can be broadly classified into two categories:

- *Theoretical methods* include deriving dominance relationships, utilization bounds, and various forms of resource augmentation factors, such as speedup factors, capacity augmentation bounds, or approximation ratios. These approaches typically give a worst-case comparison against a specific competitor, i.e., against an alternative schedulability test for the same or a different scheduling algorithm.

- *Empirical methods* include simulation of the scheduling algorithm, evaluation of the schedulability test on synthetic task sets, case studies, and experiments on real hardware. These approaches typically facilitate an average-case comparison against a number of different scheduling algorithms or schedulability tests. See [19] for a review.

In this section, theoretical methods are presented for comparing the performance of different schedulability tests and scheduling algorithms for real-time systems. The main approaches are outlined in more detail below. Note, when discussing comparisons between scheduling algorithms, we are normally referring to comparisons between exact schedulability tests for those algorithms. Comparisons are also possible using sufficient schedulability tests, thus evaluating the performance of different approximations.

- *Dominance Relationships* are used to indicate if one scheduling algorithm or schedulability test always outperforms another. For example, schedulability test $\mathcal{X}$ is said to dominate test $\mathcal{Y}$ if every task set that is schedulable according to test $\mathcal{Y}$ is also schedulable according to test $\mathcal{X}$, and there are some task sets that are schedulable according to $\mathcal{X}$ but not according to $\mathcal{Y}$. Proving a dominance relationship shows that the dominant method is always better, at least in terms of schedulability; however, no indication is given as to how good the schedulability tests (or algorithms) actually are; a dominant test may still have poor performance, just not quite as poor as that of the test that it dominates.
- *Utilization Bounds* [2, 4, 26, 40] provide a simple way of comparing different scheduling algorithms or schedulability tests. The utilization bound is the minimum total utilization of any unschedulable task set for a given scheduling algorithm and task model. Thus any task set with a total utilization $U_{sum} = \sum_{\tau_i \in \mathbf{T}} U_i$ no greater than the bound is guaranteed to be schedulable. In Chapter 4, we will prove the Liu and Layland bounds for the problem $1|spor, impl, prmp|_{\leq} D$ under the earliest-deadline-first preemptive (EDF-P) scheduler (the bound is $U_{sum} \leq 1.0$) and for the problem $1|spor, impl, prmp, fp|_{\leq} D$ under fixed-priority preemptive (FP-P) scheduling with rate-monotonic priority assignment (also called rate-monotonic (RM) scheduling) (the bound is $U_{sum} \leq \ln 2 \approx 0.693$). We note that there are also utilization-based schedulability tests that make use of task utilizations in hyperbolic or quadratic forms [7, 8, 12, 13, 28].
- *Speedup Factors* [31,46] indicate the factor ρ by which the overall speed of a system would need to be increased so that any task set that was schedulable under a reference scheduling algorithm $\mathcal{B}$ is guaranteed to be schedulable under scheduling algorithm $\mathcal{A}$. We note that the increase in speed implies that the worst-case execution time (WCET) of each task is reduced by a factor of ρ. Speedup factors illustrate the worst-case performance that one scheduling algorithm can have relative to another. If the reference algorithm $\mathcal{B}$ is an optimal scheduling algorithm or an exact schedulability test, then the quantification ρ is against the optimal result.

Speedup factors can be used to explore sub-optimality with respect to an optimal algorithm, e.g. comparing non-preemptive scheduling algorithms against EDF-P [18] or to make relative comparisons between two non-optimal algorithms, e.g. comparing fixed priority non-preemptive (FP-NP) and EDF non-preemptive (EDF-NP) scheduling [18, 50]. We note that the usefulness of speedup factors is diminished, if no schedulability test is available for the reference algorithm.

In this book, we use the negation of the above definition to quantify the failure of algorithm $\mathcal{A}$: *If $\mathcal{A}$ fails to ensure that all the task in $\mathbf{T}$ meet their deadlines, then the task set is not schedulable under the (reference) algorithm $\mathcal{B}$ when the system (i.e., each processor) is slowed down to run at speed $1/\rho$.* We note that the definition of the speedup factor always requires a reference algorithm $\mathcal{B}$. If we do not specify any reference algorithm, then we implicitly imply the comparison against an optimal scheduling algorithm or an exact schedulability test.

Specifically, for hard real-time systems, since a task set is either schedulable or not schedulable under a scheduling algorithm, it is not possible to approximation such a binary answer. The speedup factors provide quantitive metrics to quantify the imperfectness of a scheduling algorithm or a schedulability test.

- *Capacity Augmentation Bounds* [3, 38, 39] for identical (homogeneous) multiprocessor systems quantify scheduling algorithms or schedulability tests via a threshold b, such that the algorithm or test guarantees schedulability of any task set τ provided that $\max_{\tau_i \in \tau} U_i \leq \frac{1}{b}$ and $U_{sum} \leq \frac{m}{b}$, where m is the number of identical processors, and U_i is the utilization of task τ_i in task set τ with total utilization U_{sum}. The notion of capacity augmentation bounds was formally introduced in 2013 by Li et al. [38] to quantify global-EDF scheduling of task sets where each task can be further characterized using a directed acyclic graph (DAG).

- *Approximation Ratios* [25,49] for identical (homogeneous) multiprocessor systems compare the number of processors needed by (i) scheduling algorithm $\mathcal{A}$ and (ii) an optimal algorithm, to schedule any given task set, as the number of processors required by the optimal algorithm tends to infinity. The approximation ratio indicates the maximum value of the ratio of the number of processors $M_{\mathcal{A}}(\mathbf{T})$ required by algorithm $\mathcal{A}$ to schedule any given task set $\mathbf{T}$, compared to the number of processors $M_{\mathcal{O}}(\mathbf{T})$ required by an optimal algorithm, as the number of processors required by the optimal algorithm tends to infinity, i.e., $\left(\max_{\forall \mathbf{T}} \left(\frac{M_{\mathcal{A}}(\mathbf{T})}{M_{\mathcal{O}}(\mathbf{T})}\right)\right)$. Approximation ratios have been used to characterize multiprocessor scheduling.

These bounds and factors can be potentially useful to theoretically quantify the imperfectness of a scheduling algorithm or a schedulability test. Specifically, utilization bounds, speedup factors, and capacity augmentation bounds have been widely adopted and accepted by the real-time scheduling research community as the *de facto* standard theoretical tools for assessing scheduling algorithms and schedulability tests. A recent study [15] in 2017 shows that such quantitive metrics should be *handled with care*. These theoretical metrics can provide useful information; however, there are also pitfalls in their use. Problems can occur when algorithms are designed with speedup factors in mind, or conclusions are drawn taking a positive perspective solely on the basis of these results. We will discuss the potential pitfalls at the end of the book.

2.8 Anomalies and Sustainability of Analyses

A good scheduling algorithm and a good schedulability test should be **sustainable**. According to Baruah and Burns [5], "*A scheduling algorithm or a schedulability test is defined to be sustainable if any task system determined to be schedulable remains so if it behaves* ***better*** *than mandated by its system specifications.*" That is, if a sustainable scheduling algorithm derives a feasible schedule (or a sustainable schedulability test ensures the schedulability of the input), it should also guarantee the feasibility with less stringent inputs, e.g.,

- less execution time of a task,
- longer period of a periodic task,
- less number of tasks, or
- more number of processors.

An unsustainable scheduling algorithm or schedulability test may lead to scheduling anomaly. In Section 2.3.1, we already demonstrated that non-preemptive work-conserving scheduling algorithms do not behave monotonically, i.e., reducing the execution time of a job can lead to deadline misses. In addition, we provide two additional cases of such anomaly.

2.8.1 Extending Periods of Periodic Tasks with Offsets

Baruah and Burns [5] provide the following example to demonstrate that extending the periods of periodic tasks with offsets may result in unschedulability. Therefore, for such a scenario, sustainable scheduling algorithms cannot be optimal or sustainable schedulability tests cannot be exact.

▶ **Example 2.12.** Consider the following periodic real-time tasks: $\tau_1 = \{C_1 = D_1 = 1, T_1 = 2, O_1 = 0\}$ and $\tau_2 = \{C_2 = D_2 = 1, T_2 = 2, O_2 = 1\}$. This task set is perfectly schedulable under any work-conserving algorithm since the jobs generated by the two tasks have no overlap in their release times and absolute deadlines at all.

However, if we increase the period of task τ_2 from 2 to 3, the task set becomes not schedulable by any algorithm as both tasks release a job at time 4, and (at least) one of them misses the absolute deadline at time 5. ◀

2.8.2 Multiprocessor Systems with Precedence Constraints

In 1969, Graham [25] presented the scheduling anomaly for the multiprocessor scheduling problem $P_m|prec|C_{\max}$: "*changing the priority order, increasing the number of processors, reducing execution times, or weakening precedence constraints can increase the schedule length*". Here, the schedule length is the makespan and can be considered as the deadline of the jobs. This is known as the *multiprocessor anomaly*. One reason behind the multiprocessor anomaly is the non-optimality and non-sustainability of work-conserving schedules. If the scheduling algorithm is static, then such anomaly can be technically avoided.

Figure 2.8 is the input task set with precedence constraints. A job cannot be executed unless all its predecessors have finished. We assume a global scheduling algorithm that is fixed-priority non-preemptive and work-conserving. The jobs are indexed so that J_i has a higher priority than J_j if $i < j$. Figure 2.9 is the resulting global multiprocessor schedule on 3 processors. The makespan (schedule length) is 12.

Number of processors increased: When the number of processors is 4, the resulting work-conserving schedule is in Figure 2.10. The makespan is 15 since the jobs J_5, J_6, J_7, J_8 are all ready at time 2. They are executed prior to job J_9. This shows that adding more processors can be worse for the scheduling algorithm.

Reducing the execution time: When we reduce the execution time of each job by 1 unit of time, the resulting work-conserving schedule is in Fig-

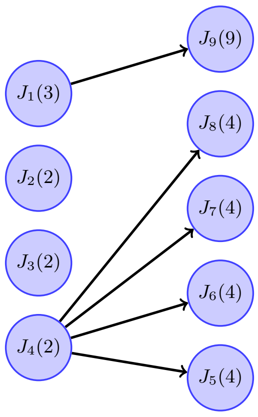


Figure 2.8: An example of jobs with precedence constraints

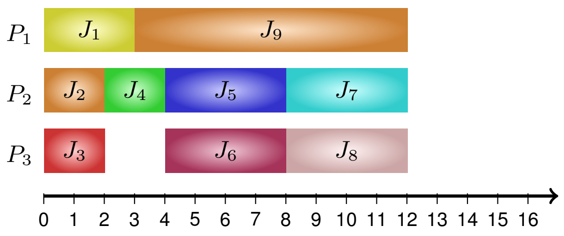


Figure 2.9: The global schedule of the jobs in Figure 2.8 on 3 processors.

ure 2.11 on 3 processors. The makespan is 13. This shows that the makespan can be worse if the execution times of some jobs are reduced.

Relaxing the precedence constraints: If we remove the precedence constraints due to J_4, the resulting work-conserving schedule is in Figure 2.12 on 3 processors. The makespan is 16. This shows that the makespan can be worse if precedence constraints are weakened.

2.8.3 Remarks

The two examples in Sections 2.8.1 and 2.8.2 in fact represent for two different sources of un-sustainability of scheduling algorithms. In the first case in Section 2.8.1, the feasibility of the task set cannot be maintained any more by extending the period of a periodic task. Although it may seem that extending the period of a periodic task makes the task set easier to be schedulable, this is not always true as demonstrated in Example 2.12. This is not a *relaxation*

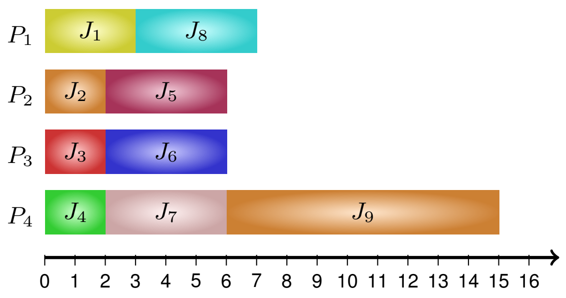


Figure 2.10: Increasing the number of processors from 3 to 4.

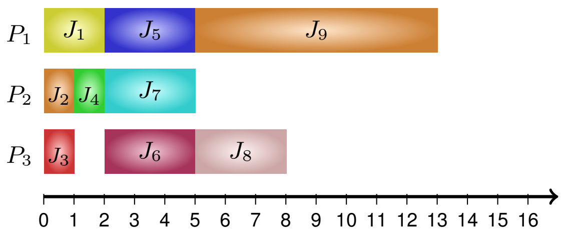


Figure 2.11: Reducing the execution time of each job by 1 unit of time, on 3 processors.

for periodic task systems. We will discuss such a relaxation for sporadic task systems in Chapter 4.

In the second case in Section 2.8.2, the scheduling algorithm itself is not optimal. Therefore, it is at a risk of non-sustainable behavior (or non-monotonic behavior with respect to the makespan). Unless there is a proof of sustainability (or monotonicity), it can be expected that such anomaly may exist. Since many multiprocessor problems are $\mathcal{NP}$-hard in the strong sense, they are usually handled with heuristic algorithms. A heuristic algorithm for such a problem may suffer from such anomaly unless the sustainability of the algorithm is proved.

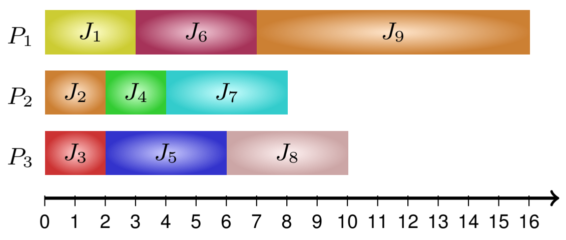


Figure 2.12: Precedence constraints weakened by removing the precedence constraints due to J_4, on 3 processors.

2.9 Exercises

▶ Exercise 2.1. For real-time systems, it is important to know the maximum (worst-case) execution time of each task a priori. What are the definition and difference between the worst-case execution time and the worst-case response time? Even if the worst-case execution time of a task is given, there are several other problems that may be encountered during the design of a scheduling algorithm for a real-time system. Can you think of some difficulties? What are possible solutions?

▶ Exercise 2.2. Suppose that the following set of jobs is given:

	J_1	J_2	J_3	J_4	J_5
r_j	0	2	8	10	15
C_j	4	3	6	3	4
d_j	6	8	16	22	20

1. What is the resulting schedule of the shortest-job-first (SJF) scheduling policy?
2. What is the resulting schedule of the earliest-deadline-first (EDF) scheduling policy?
3. What is the average response time, defined as $\frac{\sum_{j=1}^{5} f_j - r_j}{5}$, of SJF and EDF, respectively?
4. Mr. S claims that SJF is optimal for his system, and Miss E claims that EDF is optimal for her system. Is it possible that both of them are correct? Please make their descriptions more clear.

▶ Exercise 2.3. Explain sporadic real-time tasks and periodic real-time tasks and their differences. What are their typical parameters and the applications of such task models?

▶ Exercise 2.4. Explain the advantages and disadvantages of schedulability tests that are

1. necessary and sufficient
2. sufficient and sustainable
3. necessary

▶ Exercise 2.5. Explain the following problems:

1. $1|spor|RT$
2. $1|spor, prmp|RT$
3. $1|period, O_i = 0, arb|_{\leq}D$
4. $1|period, O_i = 0, impl, fp, prmp|_{\leq}D$

Part I

Uniprocessor Systems

3

Uniprocessor Aperiodic Task Systems

Uniprocessor scheduling algorithms and schedulability tests are important for various reasons. Uniprocessor systems are simpler than multiprocessor systems and a special case of more complex systems. Some results obtained for uniprocessor systems can be good algorithms or heuristics for multiprocessor systems. Furthermore, uniprocessor scheduling problems can also be important to understand the difficulty of multiprocessor scheduling problems when a multiprocessor system has a single bottleneck.

In this chapter, we will first discuss scheduling algorithms and schedulability tests for aperiodic task systems, i.e., there is no recurrence of a task. We will discuss preemptive scheduling algorithms in Sections 3.1 and 3.2. Non-preemptive scheduling algorithms will be in Section 3.3. Since an aperiodic real-time task only releases a job, we will only consider scheduling algorithms for a set of jobs $\mathbf{J}$.

3.1 Preemptive Earliest-Deadline-First (EDF-P) and Variances

The simplest scheduling problem for real-time systems is $1|d_j = D|_{\leq}D$. That is, we are supposed to schedule a set $\mathbf{J}$ of real-time jobs, in which each job arrives at time 0 and has the same (absolute) deadline D, on a uniprocessor platform. This problem is not of interest as any work-conserving (preemptive or non-preemptive) schedule would result in an optimal schedule to finish the last job at time $\sum_{J_i \in \mathbf{J}} C_i$. An exact schedulability test of any of such a work-conserving algorithm is to validate whether $\sum_{J_i \in \mathbf{J}} C_i < D$.

A slightly more complicated scheduling problem for real-time systems is $1||_{\leq}D$. That is, we are supposed to schedule a set $\mathbf{J}$ of real-time jobs, in which each job $J_i \in \mathbf{J}$ arrives at time 0 and has its absolute deadline d_i, on a uniprocessor platform. A more complicated version of this problem is $1||L_{\max}$, i.e., the objective is to minimize the maximum lateness instead of ensuring the deadline satisfaction. However, an optimal schedule for an input instance of the problem $1||L_{\max}$ that has no deadline miss is also an optimal schedule for the same input instance of the problem $1||_{\leq}D$.

A simple heuristic algorithm, called Earliest Due Date (EDD) can solve the problem $1||L_{\max}$ optimally, found by Jackson [29] in 1955. The EDD rule (also called Jackson's rule) is very simple, executes the jobs in order of non-decreasing absolute deadlines in a work-conserving manner. Since all the jobs arrive at the same time, it is unnecessary to allow preemption. The following theorem shows that EDD is optimal for the problem $1||L_{\max}$.

▶ **Theorem 3.1.** *EDD is optimal for the problem* $1||L_{max}$.

Proof. This can be proved by a simple interchange argument. Since all the jobs arrive at time 0, it is not difficult to prove that an optimal schedule should be work-conserving and non-preemptive. Let σ_A be a work-conserving schedule that does not follow the EDD rule for the input job set $\mathbf{J}$. We will show that we can transform σ_A to an EDD schedule without increasing the maximum lateness.

Since σ_A does not follow the EDD rule, in the schedule σ_A, there exist two jobs J_i and J_j in which J_i is executed *immediately* prior to J_j but $d_i > d_j$. Suppose that job J_i starts its execution at time a. Since σ_A is non-preemptive and work-conserving, $\sigma_A(t)$ is set to J_i for any $t \in (a, a + C_i]$ and $\sigma_A(t)$ is set to J_j for any $t \in (a + C_i, a + C_i + C_j]$.

We now construct another schedule σ'_A by simply swapping the execution order of J_i and J_j in σ_A. That is, σ'_A is almost the same as σ_A with an exception that J_j is executed immediately prior to J_i. The above procedure is also illustrated in Figure 3.1.

It is clear that $\sigma'_A(t) = \sigma_A(t)$ for any $t \notin (a, a+C_i+C_j]$. Therefore, the lateness of any job $J_k \in \mathbf{J} \setminus \{J_i, J_j\}$ remains the same in both schedules σ_A and σ'_A. Moreover, since J_j starts earlier in the new schedule σ'_A than in the original schedule σ_A, the lateness of J_j is reduced in the new schedule σ'_A.

We now evaluate the lateness of J_i in schedule σ'_A with two cases:

- The lateness of job J_i is less than the lateness of job J_j in schedule σ_A, i.e., $a+C_i-d_i < a+C_i+C_j-d_j$: In this case, the lateness of J_i in the new schedule σ'_A is no more than the lateness of job J_j in the original schedule σ_A since $a+C_i+C_j-d_i < a+C_i+C_j-d_j$ due to $d_i > d_j$.
- The lateness of job J_i is no less than the lateness of job J_j in schedule σ_A, i.e., $a+C_i-d_i \geq a+C_i+C_j-d_j$: This case is not possible, since $d_i > d_j$ and $C_j \geq 0$.

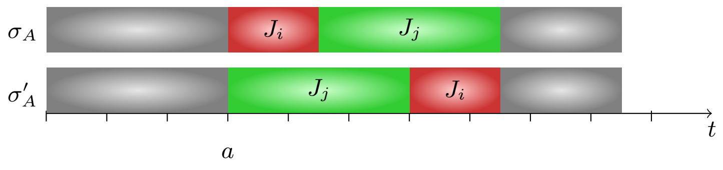


Figure 3.1: Optimality of Jackson's rule for $1||L_{\max}$.

Therefore, we reach the conclusion that the new schedule σ'_A does not increase the maximum lateness of the original schedule σ_A. We can repeat the above step until the schedule follows the EDD rule without increasing the maximum lateness. Therefore, we reach the conclusion that EDD is optimal for the problem. ◀

If the jobs have arbitrary arrival times, preemption becomes an important treatment. The problem in such a case is $1|r_j, prmp|L_{\max}$. The new algorithm was presented by Horn [1] in 1974. The algorithm, called Earliest-Deadline-First preemptive (EDF-P), schedules the arrived and unfinished job that has the earliest absolute deadline.

▶ **Theorem 3.2.** *EDF-P is optimal for the problem* $1|r_j, prmp|L_{max}$. *EDF-P is optimal for the problem* $1|r_j, prmp|_{\leq}D$.

Proof. The concept used to prove the theorem is the same as the proof of Theorem 3.1. The only additional factor here is to account for preemption. The proof is hence left as an exercise. ◀

The time complexity of EDD is $O(n \log n)$ where n is $|\mathbf{J}|$, dominated by sorting the jobs according to their absolute deadlines. The time complexity of EDF-P is also $O(n \log n)$ by applying a proper data structure, like binary heap, to insert a job to a proper priority. In EDF-P, the scheduling decision is only updated when a job arrives or completes. Here, we use a minimum binary heap (where the key of a node is the absolute deadline of a job) to illustrate the time complexity analysis. When a job completes, this job is deleted from the binary heap, which takes $O(1)$ in the amortized manner. When a job arrives, this job is inserted to the binary heap, which takes $O(\log n)$ in the amortized manner. Therefore, the time complexity to construct an EDF schedule is $O(n \log n)$.

EDF-P by design does not need to know the jobs that arrive in the future when scheduling the jobs that are ready and unfinished at time t. Even so, it

	J_1	J_2	J_3
r_j	0	4	5
C_j	10	3	10
d_j	33	28	29

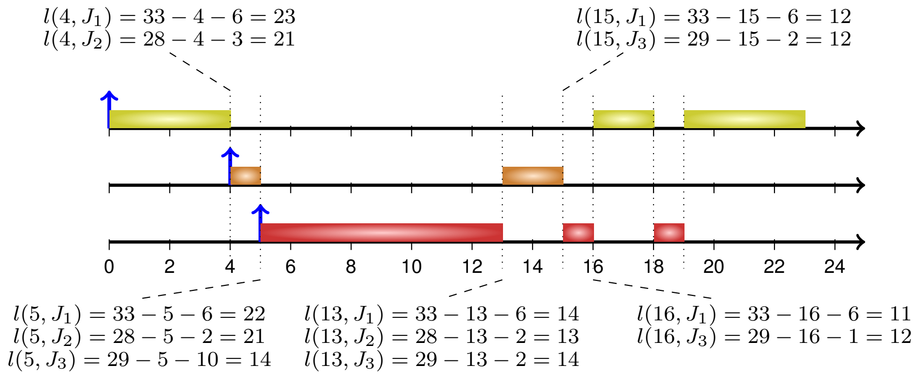


Figure 3.2: An example of least-laxity-first (LLF) scheduling

is optimal for $1|r_j, prmp|L_{max}$ and $1|r_j, prmp|_{\leq}D$. Therefore, EDF-P is in fact a clairvoyant scheduling algorithm for these two problems.

In EDF-P, the relationship of the absolute deadlines of the given jobs is translated to a priority assignment. Another possible treatment is to assign the job that has the *least laxity* the highest priority. The laxity of a job J_j at time t is defined as $d_j - t - C'_j$, where C'_j is the remaining execution time of job J_j at time t. This algorithm is called Least Laxity First (LLF), Least Slack Time First (LST), or Minimum Laxity First (MLF).

▶ **Example 3.3.** Consider to schedule the three jobs listed in the table of Figure 3.2. We assume that the scheduling decisions are taken only at discrete time points (i.e., integers) and a job is only preempted by another job with less laxity. Suppose that $l(t, J_j)$ is the laxity of job J_j at time t. At time 0, job J_1 is the only ready job in the system. Therefore, J_1 is executed in $(0, 4]$. At time 4, there are two jobs eligible to be executed. By evaluating their laxity, J_2 has less laxity and is executed in $(4, 5]$. At time 5, J_3 has the least laxity and starts its execution. According to the definition of function $l(t, J_j)$, when a job J_j is executed from $(t, t+\Delta]$, its laxity remains, and when a jobJ_j is not executed in time interval $(t, t+\Delta]$, its laxity $l(t, J_j) - l(t+\Delta, J_j) = \Delta$. Therefore, at time 13, the laxity of job J_2 becomes less than J_3 and J_2 preempts J_3. ◀

▶ **Theorem 3.4.** *LLF is optimal for the problem* $1|r_j, prmp|_{\le}D$.

Proof. This can be proved by a similar interchange argument. The proof is left as an exercise. ◀

LLF has higher time complexity and introduces more runtime overhead than EDF-P because it has to update the laxity of the jobs even without job arrivals or completions. LLF also introduces more context switches in general. Moreover, EDF-P does not need to know the execution time of the jobs, whilst LLF requires such information for scheduling decisions. This disadvantage also makes LLF difficult to be implemented in standard operating systems.

3.2 Schedulability Test of EDF-P

After presenting the EDD and EDF-P scheduling algorithms, their schedulability tests are presented in this section. Since EDD is a special case of EDF-P, we only present the proof of EDF-P formally, where the schedulability test of EDD is a simple corollary. The following theorem is due to Chetto and Silly-Chetto [17] in 1989.

▶ **Theorem 3.5.** *A set* $\mathbf{J}$ *of jobs in the problem* $1|r_j, prmp|_{\le}D$ *is schedulable under EDF-P if and only if*

$$\forall r_i < d_k, \qquad \sum_{J_j \in \mathbf{J}} C_j \mathbb{1}_{r_i \le r_j \text{ and } d_j \le d_k} \le d_k - r_i \tag{3.1}$$

Proof. **Only-if part**, i.e., the necessary schedulability test: If there exists a pair of r_i and d_k such that $r_i < d_k$ and $\sum_{J_j \in \mathbf{J}} C_j \mathbb{1}_{r_i \le r_j \text{ and } d_j \le d_k} > d_k - r_i$, then the demand of the jobs that are released and must be finished in the time interval $[r_i, d_k]$ is strictly greater than $d_k - r_i$. By the definition of a uniprocessor system in our scheduling model, at most one job is executed at a time. Therefore, the demand of the jobs that are released no earlier than r_i and must be finished no later than d_k is strictly more than the amount of available time. Therefore, one of these jobs misses its deadline no matter which uniprocessor scheduling algorithm is used.

If part, i.e., the sufficient schedulability test: We prove the condition by contrapositive. Suppose that the given set **J** of jobs is not schedulable under EDF-P for contrapositive. Let $\sigma : \mathbb{R} \to \mathbf{J} \cup \{\perp\}$ be the schedule of EDF-P for **J**, where job J_k is the first job which misses its absolute deadline d_k in schedule σ. That is,

$$\int_{r_k}^{d_k} \mathbb{1}_{\sigma(t)=J_j} dt < C_k \tag{3.2}$$

Let t_0 be the earliest instant prior to d_k, i.e., $t_0 < d_k$, such that the processor only executes jobs with absolute deadlines no later than d_k in time interval $(t_0, d_k]$ under EDF-P. That means, immediately prior to time t_0, i.e., $t = t_0 - \epsilon$ for an infinitesimal $\epsilon > 0$, $\sigma(t)$ is either $\perp$ or the job J_g with $\sigma(t) = J_g$ has absolute deadline $d_g > d_k$. We note that t_0 exists since it is at least the earliest arrival time of the jobs in **J**. Moreover, since EDF-P does not let the processor idle unless there is no job in the ready queue, $t_0 \leq r_k$.

By EDF-P, t_0 must be an arrival time of a job, called J_i. Therefore, t_0 is equal to r_i. Let $\mathbf{J}_{[r_i,d_k]}$ be the set of jobs arriving no earlier than r_i and have absolute deadlines no later than d_k. That is,

$$\mathbf{J}_{[r_i,d_k]} = \{J_j \mid r_j \geq r_i, d_j \leq d_k\}$$

By the definition of $r_i = t_0 \leq r_k$, d_k and EDF-P, the processor executes only the jobs in $\mathbf{J}_{[r_i,d_k]}$, i.e., $\sigma(t) \in \mathbf{J}_{[r_i,d_k]}$ for any $r_i < t \leq d_k$. Therefore, we know that

$$\begin{aligned}
d_k - r_i &\overset{1}{=} \left(\int_{r_i}^{d_k} \mathbb{1}_{\sigma(t)=J_k} dt\right) + \sum_{J_j \in \mathbf{J}_{[r_i,d_k]} \setminus \{J_k\}} \left(\int_{r_i}^{d_k} \mathbb{1}_{\sigma(t)=J_j} dt\right) \\
&\overset{2}{=} \left(\int_{r_k}^{d_k} \mathbb{1}_{\sigma(t)=J_k} dt\right) + \sum_{J_j \in \mathbf{J}_{[r_i,d_k]} \setminus \{J_k\}} \left(\int_{r_i}^{d_k} \mathbb{1}_{\sigma(t)=J_j} dt\right) \\
&\overset{\text{Eq. (3.2)}}{<} C_k + \sum_{J_j \in \mathbf{J}_{[r_i,d_k]} \setminus \{J_k\}} C_j \\
&= \sum_{J_j \in \mathbf{J}_{[r_i,d_k]}} C_j = \sum_{J_j \in \mathbf{J}} C_j \mathbb{1}_{r_i \leq r_j \text{ and } d_j \leq d_k}
\end{aligned}$$

where the condition $\overset{1}{=}$ is due to $\sigma(t) \in \mathbf{J}_{[r_i,d_k]}$ for any $r_i < t \leq d_k$ and the condition $\overset{2}{=}$ is due to $r_i \leq r_k$ and $\sigma(t) \neq J_k$ for $r_i < t \leq r_k$. Hence, there is a pair of r_i and d_k where $r_i < d_k$ and $\sum_{J_j \in \mathbf{J}} C_j \mathbb{1}_{r_i \leq r_j \text{ and } d_j \leq d_k} > d_k - r_i$. We reach the conclusion by contrapositive. ◀

▶ **Corollary 3.6.** *Suppose that the real-time jobs in **J** are ordered according to the absolute deadlines non-decreasingly (i.e., following the Jackson's* *rule). A set of jobs in the problem* $1||_{\leq}D$ *is schedulable under EDD if and only if*

$$\forall J_k \in \boldsymbol{J}, \qquad \sum_{i=1}^{k} C_i \leq d_k. \tag{3.3}$$

▶ **Corollary 3.7.** *The exact schedulability test for EDF-P in Theorem 3.5 is also exact for LLF.*

Proof. This is due to the optimality of LLF and EDF-P. ◀

3.3 Non-preemptive Scheduling

In this section, we will focus on non-preemptive scheduling. A schedule is **non-preemptive** if a job cannot be preempted by any other jobs, i.e., only one interval with $\sigma(t) = J_j$ for every job J_j in **J**. Whenever a job starts its execution, the processor is occupied by this job until it finishes. This restriction imposes more challenges to the design of schedulers. The following theorem shows that work-conserving is not always optimal.

▶ **Theorem 3.8.** *There exists an optimal schedule for* $1|r_j|_{\leq}D$*, which is not work-conserving.*

Proof. This can be proved by demonstrating a concrete job set that is not schedulable under any work-conserving non-preemptive scheduling algorithm but schedulable under a non-work-conserving scheduling algorithm. Consider the following two jobs: $J_1 = \{r_1 = 0, d_1 = 5, C_1 = 2\}$ and $J_2 = \{r_2 = 1, d_2 = 3, C_2 = 2\}$. A work-conserving non-preemptive scheduling algorithm executes J_1 from 0 to 2, which leads to a deadline miss of job J_2. A non-work-conserving schedule idles from 0 to 1 even when J_1 is ready at time 0, executes J_2 from time 1 to 3, and executes J_1 from 3 to 5, where both jobs can meet their deadlines. ◀

Non-preemptiveness makes the scheduler design problem much more difficult as a seemingly simple case $1|r_j|_{\leq}D$ is already $\mathcal{NP}$-complete in the strong sense. To demonstrate that, we first recall the 3-PARTITION problem [22], which is $\mathcal{NP}$-complete in the strong sense. The 3-PARTITION problem is widely used to show certain scheduling problems are $\mathcal{NP}$-hard in the strong sense.

▶ **Definition 3.9** (3-PARTITION Problem)**.** We are given a positive integer V, a positive integer M, and a set of $3M$ integer numbers $\{v_1, v_2, \ldots, v_{3M}\}$

with the condition $\sum_{i=1}^{3M} v_i = MV$, in which $V/4 < v_i < V/2$ and $M \geq 3$. Therefore, $V \geq 3$.
Objective: The problem is to partition the given $3M$ integer numbers into M disjoint sets $\mathbf{V}_1, \mathbf{V}_2, \ldots, \mathbf{V}_M$ such that the sum of the numbers in each set $\mathbf{V}_i$ for $i = 1, 2, \ldots, M$ is V, i.e., $\sum_{v_j \in \mathbf{V}_i} v_j = V$. ◀

▶ **Theorem 3.10.** *The problem $1|r_j|_{\leq}D$ is $\mathcal{NP}$-complete in the strong sense.*

Proof. The problem is in $\mathcal{NP}$ since validating the feasibility of the schedule takes polynomial time. The completeness is due to a polynomial-time reduction from the 3-PARTITION problem [22] to the problem $1|r_j|_{\leq}D$.

From an input instance of the 3-PARTITION problem, we create $n = 4M - 1$ jobs, defined as follows:

$$\begin{aligned} &r_j = jV + j - 1, C_j = 1, d_j = jV + j, && j = 1, \ldots, M-1 \\ &r_j = 0, C_j = v_{j-M+1}, d_j = MV + M - 1, && j = M, \ldots, 4M-1 \end{aligned}$$

This reduction takes polynomial time. For $j = 1, \ldots, M-1$, the construction of the jobs leads to $d_j - r_j - C_j = 0$. Therefore, to meet their deadlines, they have to be executed immediately when they arrive. As a result, the remaining $3M$ jobs can only be executed in M time intervals $(0, V]$, $(V+1, 2V+1], \ldots,$ $(MV - V + M - 1, MV + M - 1]$, each with a length of V. This can be feasibly done if and only if the remaining $3M$ jobs can be partitioned over these M time intervals each with a length of V, which can be done if and only if the input instance of the 3-PARTITION problem has a solution. ◀

▶ **Corollary 3.11.** *The problem $1|r_j|L_{\max}$ is $\mathcal{NP}$-hard in the strong sense.*

The problems $1|r_j|L_{\max}$ and $1|r_j|_{\leq}D$ are important because they are often subproblems in more complicated multiprocessor scheduling problems. There have been considerable amount of work that has resulted in several branch-and-bound methods, e.g., [11, 41, 45] and effective heuristic and approximation algorithms, e.g., [27, 48].

It should be first noted that the problem $1|r_j|L_{\max}$ cannot be approximated with a bounded approximation ratio because the optimal schedule may have no lateness at all and any approximation leads to an unbounded approximation ratio. However, a variance of this problem can be easily approximated. This is known as the *delivery-time* model of the problem $1|r_j|L_{\max}$. In this model, each job J_j has its release time r_j, processing time C_j, and delivery time $q_j \geq 0$. After a job finishes its execution on a processor, its result (final product) needs q_j amount of time to be delivered to the customer.

Therefore, the result of job J_j is *delivered* at time $f_j + q_j$, where f_j is the finishing time of job J_j. This is identical to the scenario as if we set the absolute deadline d_j of J_j to $-q_j$. Minimizing the maximum lateness in this model is equivalent to the minimization of the maximum delivery time, i.e., $\max_{J_j \in \mathbf{J}} f_j - d_j = \max_{J_j \in \mathbf{J}} f_j + q_j$.

The delivery-time model of the problem $1|r_j|L_{\max}$ can then be effectively approximated. The **extended Jackson's rule** is as follows: "Whenever the processor is free and one or more jobs is available for processing, schedule an available job with largest delivery time. " Recall that the absolute deadline of job J_j in this case is $-q_j$.

▶ **Theorem 3.12.** *The extended Jackson's rule is a polynomial-time* 2*-approximation algorithm for the delivery-time model of the problem* $1|r_j|L_{\max}$.

Proof. A sketched proof can be found in [27]. ◀

Potts [48] observed some nice properties when the extended Jackson's rule is applied. Suppose that the last delivery is due to a job J_c. Let J_a be the earliest scheduled job so that the processor in the problem $1|r_j|L_{\max}$ is not idle between the processing of J_a and J_c. The sequence of the jobs that are executed sequentially from $J_a, \ldots,$ to J_c is called a *critical sequence*. By the definition of J_a, all jobs in the critical sequence must be released no earlier than the release time r_a of job J_a. If the delivery time of any job in the critical sequence is not shorter than the delivery time q_c of J_c, then it can be proved that the extended Jackson's rule is optimal for the problem $1|r_j|L_{\max}$. However, if the delivery time q_b of a job J_b in the critical sequence is shorter than the delivery time q_c of J_c, the extended Jackson's rule may start a non-preemptive job J_b too early. Such a job J_b that appears last in the critical sequence is called the *interference job* of the critical sequence.

Potts [48] suggested to *attempt at improving the schedule by forcing some interference job to be executed after the critical job* J_c, i.e., by delaying the release time of J_b from r_b to $r'_b = r_c$. This procedure is repeated for at most n iterations and the best schedule among the iterations is returned as the solution.

▶ **Theorem 3.13.** *Potts' iterative proces is a polynomial-time* 1.5*-approximation algorithm for the delivery-time model of the problem* $1|r_j|L_{\max}$.

Proof. The proof can be found in [27]. ◀

Hall and Shmoys [27] further improved the approximation ratio to $4/3$ by handling a special case when there are two jobs J_i and J_h with $C_i >$

$\sum_{J_j \in \mathbf{J}} C_j/3$ and $C_h > \sum_{J_j \in \mathbf{J}} C_j/3$ and running Potts' algorithm for $2n$ iterations.[1]

▶ **Theorem 3.14.** *Algorithm* ***HS*** *is a polynomial-time $4/3$-approximation algorithm for the delivery-time model of the problem $1|r_j|L_{\max}$.*

Proof. The proof can be found in [27]. ◀

[1] Hall and Shmoys [27] further use the concept of forward and inverse problems of the input instance of $1|r_j|L_{\max}$.

3.4 Exercises

▶ Exercise 3.1. For real-time systems, it is important to know the maximum (worst-case) execution time of each task a priori. What are the definition and difference between the worst-case execution time and the worst-case response time? Even if the worst-case execution time of a task is given, there are several other problems that may be encountered during the design of a scheduling algorithm for a real-time system. Can you think of some difficulties? What are possible solutions?

▶ Exercise 3.2. Complete the proof of Theorem 3.2.

▶ Exercise 3.3. Complete the proof of Theorem 3.4.

▶ Exercise 3.4. Use the example in Figure 3.2 to compare the overhead of LLF and EDF-P, including the number of context switches and the number of time points to make scheduling decisions.

4

Uniprocessor Periodic and Sporadic Task Systems

We consider periodic and sporadic task systems on a uniprocessor platform in this chapter, starting from the well-known EDF scheduling algorithm in preemptive and non-preemptive settings in Sections 4.1 and 4.2 respectively. We will then discuss the fixed-priority scheduling algorithms in preemptive and non-preemptive settings in Sections 4.3 and 4.4 respectively. These classical results are widely used and applied as basic subroutines in real-time and embedded systems.

4.1 Preemptive Dynamic-Priority Scheduling

For the preemptive EDF (EDF-P) scheduling algorithm, the job in the ready queue whose absolute deadline is the earliest is executed on the processor. Since the absolute deadline of a job released by a periodic/sporadic task is well-defined when it arrives to the system, the EDF-P scheduling algorithm, presented in Section 3.1, can always be applied if the relative deadline of a task is defined. The scheduling algorithm EDF-P itself does not need any information from the tasks' perspective. As shown in Section 3.1, EDF-P is an optimal scheduling algorithm for the problem $1|prmp, r_j|_{\leq}D$ even in the online setting, where the arrival times of the jobs are only known when they are released to the system. Therefore, EDF-P is also an optimal scheduling algorithm for the problems $1|prmp, spor, arb|_{\leq}D$ and $1|prmp, period, arb|_{\leq}D$.

The key issue is to validate the schedulability of EDF-P. Towards this, the *demand bound function* $\text{DBF}_i(t)$, defined by Baruah et al. [6], has been widely used to specify the maximum demand of task τ_i to be released and finished in a time interval with length equal to t:

$$\text{DBF}_i(t) = \max\left\{0, \left\lfloor \frac{t - D_i}{T_i} \right\rfloor + 1\right\} \times C_i \tag{4.1}$$

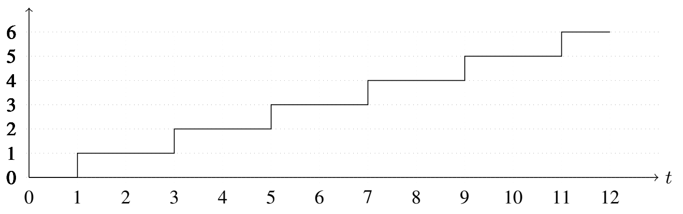


Figure 4.1: An example of the demand bound function $\text{DBF}_i(t)$ for task $\tau_i = \{T_i = 2, C_i = 1, D_i = 1\}$.

Figure 4.1 demonstrates an example when $\tau_i = \{T_i = 2, C_i = 1, D_i = 1\}$.

To prove the correctness of such a demand bound function, we focus on all possible feasible sets of jobs generated by a sporadic/periodic real-time task τ_i. Recall that a feasible set $\mathbf{FJ}_i$ of jobs generated by a sporadic/periodic real-time task τ_i should satisfy the following conditions:

- The actual execution time $C_{i,j}$ of job $J_{i,j}$ satisfies $C_{i,j} \leq C_i$.
- $d_{i,j} = r_{i,j} + D_i$ for any integer j with $j \geq 1$.
- $r_{i,j} \geq r_{i,j-1} + T_i$ for any integer j with $j \geq 2$.

▶ **Lemma 4.1.** *For a given feasible set $\boldsymbol{FJ}_i$ of jobs generated by a sporadic/periodic real-time task τ_i, let $\boldsymbol{FJ}_{i,[r,r+t]}$ be the subset of the jobs in $\boldsymbol{FJ}_i$ arriving no earlier than r and have absolute deadlines no later than $r + t$. That is,*

$$\boldsymbol{FJ}_{i,[r,r+t]} = \{J_{i,j} \mid J_{i,j} \in \boldsymbol{FJ}_i, r_{i,j} \geq r, d_{i,j} \leq r + t\} \tag{4.2}$$

For any r and any $t > 0$,

$$\sum_{J_{i,j} \in \boldsymbol{FJ}_{i,[r,r+t]}} C_{i,j} \leq \text{DBF}_i(t). \tag{4.3}$$

Proof. By definition, $\text{DBF}_i(t) \geq 0$. Therefore, if $\mathbf{FJ}_{i,[r,r+t]}$ is an empty set, we reach the conclusion.

We consider that $\mathbf{FJ}_{i,[r,r+t]}$ is not empty for the rest of the proof. Let $J_{i,j*}$ be the first job generated by task τ_i in $\mathbf{FJ}_{i,[r,r+t]}$. By the definition of $\mathbf{FJ}_{i,[r,r+t]}$ in Eq. (4.2), the arrival time r_{i,j^*} of job J_{i,j^*} is no less than r, i.e., $r_{i,j^*} \geq r$. Since $\mathbf{FJ}_{i,[r,r+t]}$ is not empty, $r_{i,j^*} + D_i \leq r + t$.

Since $r_{i,j} \geq r_{i,j-1} + T_i$ for any integer j with $j \geq 2$ for the jobs in $\mathbf{FJ}_i$ as well as the jobs in $\mathbf{FJ}_{i,[r,r+t]}$, the absolute deadlines of the *subsequent* jobs in $\mathbf{FJ}_{i,[r,r+t]}$ are *at least* $r_{i,j^*} + T_i + D_i, r_{i,j^*} + 2T_i + D_i, r_{i,j^*} + 3T_i + D_i, \ldots$. Therefore, there are at most $\left\lfloor \frac{r+t-(r_{i,j^*}+D_i)}{T_i} \right\rfloor + 1 \leq \left\lfloor \frac{t-D_i}{T_i} \right\rfloor + 1$ jobs in $\mathbf{FJ}_{i,[r,r+t]}$ since $r \leq r_{i,j^*}$. Since the actual execution time $C_{i,j}$ of each job $J_{i,j}$ is no more than C_i by the definition of the jobs in $\mathbf{FJ}_i$, we reach the conclusion. ◀

With the help of Lemma 4.1, the following theorem follows directly from Theorem 3.5 by Baruah et al. [6] with some modification.

▶ **Theorem 4.2.** *A set* $\boldsymbol{T}$ *of sporadic tasks in the problem* $1|spor, prmp|_{\leq}D$ *is schedulable under EDF-P* ***if and only if***

$$\forall t > 0, \qquad \sum_{\tau_i \in \boldsymbol{T}} \text{DBF}_i(t) \leq t \tag{4.4}$$

Proof. Only-if part, i.e., the necessary schedulability test. We prove the condition by contrapositive. Suppose that there exists a $t > 0$ such that $\sum_{\tau_i \in \mathbf{T}} \text{DBF}_i(t) > t$, for contrapositive.

For each task τ_i, we create a feasible set of jobs generated by task τ_i by releasing the jobs periodically starting from time 0, and their actual execution times are all set to C_i. By the definition of a uniprocessor system in our scheduling model, at most one job is executed at a time. Therefore, the demand of the jobs that are released no earlier than 0 and must be finished no later than t is strictly more than the amount of available time since $\sum_{\tau_i \in \mathbf{T}} \text{DBF}_i(t) > t$. Therefore, (at least) one of these jobs misses its deadline no matter which uniprocessor scheduling algorithm is used.

Therefore, we can conclude that if the task set $\mathbf{T}$ is schedulable under EDF-P, then $\sum_{\tau_i \in \mathbf{T}} \text{DBF}_i(t) \leq t, \forall t > 0$.

If part, i.e., the sufficient schedulability test: We prove the condition by contrapositive. Suppose that the given task set $\mathbf{T}$ is not schedulable under EDF-P for contrapositive.

Then, there exists a feasible collection of jobs generated by $\mathbf{T}$ which cannot be feasibly scheduled under EDF-P. Let $\mathbf{FJ}$ be such a collection of jobs, where $\mathbf{FJ}_i$ is its subset generated by a sporadic real-time task τ_i in $\mathbf{T}$. Let $\sigma : \mathbb{R} \rightarrow \mathbf{FJ} \cup \{\perp\}$ be the schedule of EDF-P for $\mathbf{FJ}$. Since at least one job misses its deadline in σ, let job $J_{k,\ell}$ be the first job which misses its absolute deadline $d_{k,\ell}$ in schedule σ. That is,

$$\int_{r_{k,\ell}}^{d_{k,\ell}} \mathbb{1}_{\sigma(t)=J_{k,\ell}} dt < C_{k,\ell} \leq C_k \tag{4.5}$$

Similar to the proof of Theorem 3.5, let t_0 be the earliest instant prior to $d_{k,\ell}$, i.e., $t_0 < d_{k,\ell}$, such that the processor only executes jobs with absolute deadlines no later than $d_{k,\ell}$ in time interval $(t_0, d_{k,\ell}]$ under EDF-P. That means, immediately prior to time t_0, i.e., $t = t_0 - \epsilon$ for an infinitesimal ϵ, $\sigma(t)$ is either $\perp$ or a job whose absolute deadline is (strictly) greater than $d_{k,\ell}$. We note that t_0 exists since it is at least the earliest arrival time of the jobs in $\mathbf{FJ}$. Moreover, since EDF-P does not let the processor idle unless there is no job in the ready queue, $t_0 \leq r_{k,\ell}$.

Let $\mathbf{FJ}_{i,[t_0,d_{k,\ell}]}$ be the subset of the jobs in $\mathbf{FJ}_i$ arriving no earlier than t_0 and have absolute deadlines no later than $d_{k,\ell}$. That is, we define $\mathbf{FJ}_{i,[t_0,d_{k,\ell}]}$ by setting r to t_0 and t to $d_{k,\ell} - t_0$ in Eq. (4.2). Let $\mathbf{FJ}_{[t_0,d_{k,\ell}]}$ be $\cup_{\tau_i \in \mathbf{T}} \mathbf{FJ}_{i,[t_0,d_{k,\ell}]}$ for notational brevity.

By the definition of t_0, $d_{k,\ell}$ and EDF-P, the processor executes only the jobs in $\mathbf{FJ}_{[t_0,d_{k,\ell}]}$, i.e., $\sigma(t) \in \mathbf{FJ}_{[t_0,d_{k,\ell}]}$ for any $t_0 < t \leq d_{k,\ell}$. Therefore,

$$\begin{aligned}
d_{k,\ell} - t_0 &\overset{1}{=} \left(\int_{t_0}^{d_{k,\ell}} \mathbb{1}_{\sigma(t)=J_{k,\ell}} dt\right) + \sum_{J_{i,j} \in \mathbf{FJ}_{[t_0,d_{k,\ell}]} \setminus \{J_{k,\ell}\}} \left(\int_{t_0}^{d_{k,\ell}} \mathbb{1}_{\sigma(t)=J_{i,j}} dt\right) \\
&\overset{2}{\leq} \left(\int_{t_0}^{d_{k,\ell}} \mathbb{1}_{\sigma(t)=J_{k,\ell}} dt\right) + \left(\sum_{\tau_i \in \mathbf{T}} \sum_{J_{i,j} \in \mathbf{FJ}_{i,[t_0,d_{k,\ell}]}} C_{i,j}\right) - C_{k,\ell} \\
&\overset{3}{=} \left(\int_{r_{k,\ell}}^{d_{k,\ell}} \mathbb{1}_{\sigma(t)=J_{k,\ell}} dt\right) + \left(\sum_{\tau_i \in \mathbf{T}} \sum_{J_{i,j} \in \mathbf{FJ}_{i,[t_0,d_{k,\ell}]}} C_{i,j}\right) - C_{k,\ell} \\
&\overset{\text{Eq. (4.5)}}{<} C_{k,\ell} + \left(\sum_{\tau_i \in \mathbf{T}} \sum_{J_{i,j} \in \mathbf{FJ}_{i,[t_0,d_{k,\ell}]}} C_{i,j}\right) - C_{k,\ell} \\
&\overset{\text{Eq. (4.3)}}{\leq} \sum_{\tau_i \in \mathbf{T}} \mathrm{DBF}_i(d_{k,\ell} - t_0)
\end{aligned}$$

where the condition $\overset{1}{=}$ is due to $\sigma(t) \in \mathbf{FJ}_{[t_0,d_{k,\ell}]}$ for any $t_0 < t \leq d_{k,\ell}$, the condition $\overset{2}{\leq}$ is due to the definition of a schedule of the jobs in $\mathbf{FJ}_{[t_0,d_{k,\ell}]} \setminus \{J_{k,\ell}\}$, the condition $\overset{3}{=}$ is due to $t_0 \leq r_{k,\ell}$ and $\sigma(t) \neq J_{k,\ell}$ for $t_0 < t \leq r_{k,\ell}$.

Hence, there is a certain $\Delta = d_{k,\ell} - t_0$ with $\sum_{\tau_i \in \mathbf{T}} \text{DBF}_i(\Delta) > \Delta$. We reach our conclusion by contrapositive. ◀

▶ **Corollary 4.3.** *A set **T** of periodic tasks is schedulable under EDF-P for the problem* $1|period, O_i = 0, prmp|_{\leq}D$ ***if and only if*** *Eq. (4.4) holds.*

Proof. It follows directly from the proof of Theorem 4.2. ◀

▶ **Corollary 4.4.** *A set **T** of periodic tasks is schedulable under EDF-P for the problem* $1|period, prmp|_{\leq}D$ *if Eq. (4.4) holds.*

Proof. It follows directly from the if-part proof of Theorem 4.2. ◀

Note that the schedulability test in Corollary 4.4 is only a sufficient schedulability test for the problem $1|period, prmp|_{\leq}D$. For periodic task sets, it is possible that the test $\sum_{\tau_i \in \mathbf{T}} \text{DBF}_i(t) \leq t, \quad \forall t > 0$ is pessimistic if it is not possible construct a concrete feasible collection of jobs generated by the given tasks in **T** to release their first jobs at the same time. More precisely, the only-if-part proof in Theorem 4.2 cannot be used for the problem $1|period, prmp|_{\leq}D$ because the task set may not permit such a feasible collection of jobs to be generated by **T**.

▶ **Example 4.5.** Consider the following example with two tasks in **T**, where $C_1 = 2, D_1 = 2, T_1 = 10, O_1 = 0$ and $C_2 = 2, D_2 = 2, T_2 = 10, O_1 = 2$. In this example, $\text{DBF}_1(2) = 2$ and $\text{DBF}_2(2) = 2$. Therefore, $\text{DBF}_1(2) + \text{DBF}_2(2) = 4 > 2$, which implies that the task set is (potentially) not schedulable under EDF-P based on the schedulability test in Corollary 4.4. However, this task set is in fact schedulable under EDF-P since the jobs released by the two tasks are perfectly separate from each other. The schedulability of the task set can be easily validated by simulating the schedule from time 0 to 12. ◀

For constrained- and arbitrary-deadline task sets, testing the schedulability of EDF-P for the problem $1|period, prmp|_{\leq}D$ is in fact $\mathcal{NP}$-complete in the strong sense [6, 9]. Specifically, Leung and Merrill [36] provide a polynomial-time reduction from the Simultaneous Congruences problem to the problem $1|period, prmp|_{\leq}D$. The Simultaneous Congruences problem is proved to be $\mathcal{NP}$-complete by Leung and Whitehead [37]. Baruah et al. [6] later show that the Simultaneous Congruences problem is $\mathcal{NP}$-complete in the strong sense. Therefore, testing the schedulability of $1|period, prmp|_{\leq}D$ is $\mathcal{NP}$-complete in the strong sense.

However, for implicit-deadline periodic or sporadic task systems, we can have the following exact schedulability test of EDF-P by Liu and Layland [40]:

▶ **Theorem 4.6.** *A set* ***T*** *of tasks is schedulable under EDF-P for the problem* $1|spor, prmp, impl|_{\leq}D$ *or* $1|period, prmp, impl|_{\leq}D$ ***if and only if***

$$\sum_{\tau_i \in \boldsymbol{T}} U_i \leq 1 \tag{4.6}$$

Proof. **Only-if part**, i.e., the necessary schedulability test. This can be easily proved as the system is *overloaded* in the worst case. If $\sum_{\tau_i \in \mathbf{T}} U_i > 1$, it is not difficult to prove that there exists a feasible collection of jobs generated by **T**, in which at least one job misses its deadline. This holds for both periodic and sporadic task sets.

If part, i.e., the sufficient schedulability test. For an implicit-deadline task, due to $D_i = T_i$, we have $\text{DBF}_i(t) = \max\left\{0, \left\lfloor \frac{t-T_i}{T_i} \right\rfloor + 1\right\} \times C_i = \left\lfloor \frac{t}{T_i} \right\rfloor C_i$ for any $t > 0$. Therefore, for any $t > 0$,

$$\sum_{\tau_i \in \mathbf{T}} \text{DBF}_i(t) = \sum_{\tau_i \in \mathbf{T}} \left\lfloor \frac{t}{T_i} \right\rfloor C_i \leq \sum_{\tau_i \in \mathbf{T}} \frac{t}{T_i} C_i = t \cdot \sum_{\tau_i \in \mathbf{T}} U_i \leq t \tag{4.7}$$

By Theorem 4.2, we reach the conclusion. ◀

The exact schedulability test in Theorem 4.6 shows that there is no loss of utilization for implicit-deadline task systems under EDF-P.

▶ **Corollary 4.7.** *A set* ***T*** *of tasks with* $D_i \geq T_i$ *for every task* τ_i *in* ***T*** *for the problem* $1|spor, prmp|_{\leq}D$ *or* $1|period, prmp|_{\leq}D$ *is schedulable under EDF-P* ***if and only if*** $\sum_{\tau_i \in \boldsymbol{T}} U_i \leq 1$.

Proof. The proof is identical to the proof in Theorem 4.6. The only difference is that for $t > 0$ when $D_i \geq T_i$, we have

$$\text{DBF}_i(t) = \max\left\{0, \left\lfloor \frac{t-D_i}{T_i} \right\rfloor + 1\right\} \times C_i \leq \max\left\{0, \left\lfloor \frac{t-T_i}{T_i} \right\rfloor + 1\right\} \times C_i = \left\lfloor \frac{t}{T_i} \right\rfloor C_i$$

◀

With the above analyses, the sufficient schedulability tests and their exactness are summarized in Table 4.1.

Time Complexity

Testing whether the schedulability condition of Eq. (4.4) in Theorem 4.2 in fact needs exponential time complexity. We will discuss the implementation of such a test in Section 4.5.

	Relative deadline	sufficient test	exact?
Sporadic or Periodic with the same offset	Implicit	Theorem 4.6	yes
	constrained	Theorem 4.2	yes
	arbitrary	Theorem 4.2	yes
Periodic	Implicit	Theorem 4.6	yes
	constrained	Corollary 4.4	no
	arbitrary	Corollary 4.4	no

Table 4.1: Sufficient schedulability tests of EDF-P and their exactness.

4.2 Non-Preemptive Dynamic-Priority Scheduling

This section considers non-preemptive uniprocessor scheduling algorithms for periodic and sporadic real-time task systems. One of the well-known approaches is the non-preemptive EDF (EDF-NP) scheduling algorithm. Under EDF-NP, the job in the ready queue whose absolute deadline is the earliest is executed *non-preemptively* on the processor. As already shown in Theorem Section 3.3, EDF-NP is not an optimal scheduling algorithm for the problem $1|r_j|L_{\max}$ even in the offline setting, where the arrival times of the jobs are known at beginning. The optimal schedule for the problem $1|r_j|L_{\max}$ may not be work-conserving.

The following theorem presents the exact schedulability test of EDF-NP for sporadic real-time tasks.

▶ **Theorem 4.8.** *A set **T** of sporadic tasks in the problem $1|spor|_{\leq}D$ is schedulable under EDF-NP **if and only if***

$$\forall t > 0, \qquad \left(\sum_{\tau_i \in \boldsymbol{T}} \texttt{DBF}_i(t) \right) + \max_{\tau_q : D_q > t} \{C_q - \epsilon\} \leq t \tag{4.8}$$

where $\epsilon > 0$ is infinitesimal.

Proof. Only-if part, i.e., the necessary schedulability test. We prove the condition by contrapositive. Suppose that there exists a $t > 0$ such that $\sum_{\tau_i \in \mathbf{T}} \texttt{DBF}_i(t) + \max_{\tau_q : D_q > t}\{C_q - \epsilon\} > t$, for contrapositive. There are two cases. First, if $D_q \leq t$ for all $\tau_q \in \mathbf{T}$, then this is the same as the proof of the necessary schedulability test in the proof of Theorem 4.2. Therefore, we only consider that there is a task τ_q with $D_q > t$. Then, we release a job of task τ_q at time $-\epsilon$ and release the first jobs of the other tasks $\mathbf{T} \setminus \{\tau_q\}$ at time 0 and their subsequent jobs periodically. All jobs use their corresponding worst-case execution times. Due to the work-conserving execution property of the EDF-NP, the job of task τ_q is executed from $-\epsilon$ to $C_q - \epsilon$. The demand of the jobs that are released no earlier than 0 and must be finished no later than t is strictly more than the amount of remaining available time $t - C_q + \epsilon$ since $\sum_{\tau_i \in \mathbf{T}} \texttt{DBF}_i(t) > t - C_q + \epsilon$. Therefore, (at least) one of these jobs misses its deadline no matter which uniprocessor *non-preemptive* scheduling algorithm is used from time $C_q - \epsilon$ to time t.

If part, i.e., the sufficient schedulability test: We prove the condition by contrapositive. Suppose that the given task set **T** is not schedulable under EDF-NP for contrapositive.

Then, there exists a feasible collection of jobs generated by **T** which cannot be feasibly scheduled under EDF-NP. Let **FJ** be such a collection of jobs, where $\mathbf{FJ}_i$ is its subset generated by a sporadic/periodic real-time task τ_i in **T**. Let $\sigma : \mathbb{R} \rightarrow \mathbf{FJ} \cup \{\perp\}$ be the schedule of EDF-NP for **FJ**. Since at least one job misses its deadline in σ, let job $J_{k,\ell}$ be the first job which misses its absolute deadline $d_{k,\ell}$ in the EDF-NP schedule σ. That is,

$$\int_{r_{k,\ell}}^{d_{k,\ell}} \mathbb{1}_{\sigma(t)=J_{k,\ell}} dt < C_{k,\ell} \leq C_k \tag{4.9}$$

Let t_0' be the earliest instant *no later than* $d_{k,\ell}$, i.e., $t_0' \leq d_{k,\ell}$, such that the processor only executes jobs with absolute deadlines no later than $d_{k,\ell}$ in time interval $(t_0', d_{k,\ell}]$ under EDF-NP. That means, immediately prior to time t_0', i.e., $t = t_0' - \epsilon$ for an infinitesimal $\epsilon > 0$, $\sigma(t)$ is either $\perp$ or a job whose absolute deadline is (strictly) greater than $d_{k,\ell}$. The definition of t_0' is slightly different from the definition of t_0 in the proof of Theorem 4.2, because it is possible that the job executed immediately prior to $d_{k,\ell}$ in schedule σ is a job started before $r_{k,\ell}$ under EDF-NP. We note that t_0' exists since it is at least the earliest arrival time of the jobs in **FJ** or $d_{k,\ell}$.

The reader may notice that the above construction is almost the same as the proof of Theorem 4.2. There are two cases to consider. **Case 1:** $\sigma(t_0' - \epsilon)$ is $\perp$. This is an easier case, since the remaining part of the proof is the same as the rest of the proof of Theorem 4.2. For this case, $\Delta < \sum_{\tau_i \in \mathbf{T}} \texttt{DBF}_i(\Delta)$, where $\Delta = d_{k,\ell} - t_0' \geq d_{k,\ell} - r_{k,\ell} > 0$ since $t_0' \leq r_{k,\ell}$ for such a case.

For the rest of the proof, we focus on **Case 2:** $\sigma(t_0' - \epsilon)$ is a job $J_{q,p}$ (i.e., a job of task τ_q) whose absolute deadline is strictly greater than $d_{k,\ell}$. Since the schedule σ is a non-preemptive schedule, job $J_{q,p}$ must be consecutively executed from a certain time t_0 to t_0' and $t_0' - t_0$ is, therefore, *at most* the actual execution time $C_{q,p}$ of job $J_{q,p}$. According to EDF-NP, job $J_{q,p}$ starts its execution at time t_0 because its absolute deadline is the earliest among the jobs in the ready queue. Since we define t'_0 such that the processor only executes jobs with absolute deadlines no later than $d_{k,\ell}$ in time interval $(t'_0, d_{k,\ell}]$, those jobs must not be ready prior to t_0, i.e., they arrive *strictly* later than t_0. Otherwise, EDF-NP would not start to execute job $J_{q,p}$ at time t_0.

Let $\mathbf{FJ}_{i,[t_0+\epsilon,d_{k,\ell}]}$ be the subset of the jobs in $\mathbf{FJ}_i$ arriving no earlier than $t_0 + \epsilon$ and have absolute deadlines no later than $d_{k,\ell}$. That is, we define $\mathbf{FJ}_{i,[t_0+\epsilon,d_{k,\ell}]}$ by setting r to $t_0 + \epsilon$ and t to $d_{k,\ell} - t_0 - \epsilon$ in Eq. (4.2). Let $\mathbf{FJ}_{[t_0+\epsilon,d_{k,\ell}]}$ be $\cup_{\tau_i \in \mathbf{T}} \mathbf{FJ}_{i,[t_0+\epsilon,d_{k,\ell}]}$ for notational brevity.

Therefore, the EDF-NP schedule σ is busy executing jobs in $\{J_{q,p}\} \cup \mathbf{FJ}_{[t_0+\epsilon,d_{k,\ell}]}$ in time interval $(t_0, d_{k,\ell}]$ and job $J_{k,\ell}$ misses its deadline. By definition, $J_{q,p} \notin \mathbf{FJ}_{[t_0+\epsilon,d_{k,\ell}]}$. Moreover, since $r_{q,p} \leq t_0$ and $d_{q,p} > d_{k,\ell}$, the relative deadline D_q of task τ_q is $D_q = d_{q,p} - r_{q,p} > d_{k,\ell} - t_0 > d_{k,\ell} - t_0 - \epsilon$.

Let Δ be $d_{k,\ell} - t_0 - \epsilon$ for brevity. Since $d_{k,\ell} \geq t'_0 > t_0 + \epsilon$ by definition, $\Delta > 0$. By adopting the inequality in Eq. (4.9) and similar steps in the last part of the proof of Theorem 4.2 (*left as an exercise*), it can be proved that

$$\begin{aligned}\Delta <& C_{k,\ell} + \left(\sum_{\tau_i \in \mathbf{T}} \sum_{J_{i,j} \in \mathbf{FJ}_{i,[t_0+\epsilon,d_{k,\ell}]}} C_{i,j} \right) - C_{k,\ell} + (C_{q,p} - \epsilon) \\ \leq& \left(\sum_{\tau_i \in \mathbf{T}} \mathrm{DBF}_i(\Delta) \right) + \max_{\tau_q : D_q > \Delta} \{C_q - \epsilon\}\end{aligned}$$

We reach the conclusion by contrapositive. ◀

The schedulability test in Theorem 4.8 is slightly different from the original test by George et al. [23]. In Theorem 4.8, the execution of a job is in a continuous time domain, whereas the test by George et al. [23] assumes a discrete time domain, where the job arrival times and job execution times are (positive) integers.

▶ **Theorem 4.9.** *A set **T** of sporadic tasks in the problem $1|spor|_{\leq}D$, where the job arrival times and job execution times are integers, is schedulable under EDF-NP **if and only if***

$$\forall t > 0, \qquad \left(\sum_{\tau_i \in \boldsymbol{T}} \mathrm{DBF}_i(t) \right) + \max_{\tau_q : D_q > t} \{C_q - 1\} \leq t \tag{4.10}$$

Proof. The proof is identical to that in Theorem 4.8 in a discrete time domain. ◀

A similar result was also presented by Jeffay et al. [30] for the problem $1|spor, impl|_{\leq}D$ under a discrete time domain. However, Jeffay et al. [30] concluded that the same schedulability test is *exact* also for the problem $1|period|_{\leq}D$ when each periodic task has a concrete offset. This is unfortunately only a sufficient test, which can be shown by using a similar example presented in Example 4.5.

▶ **Corollary 4.10.** *A set **T** of periodic tasks is schedulable under EDF-NP for the problem $1|period|_{\leq}D$ if Eq. (4.8) holds.*

Proof. It follows directly from the if-part proof of Theorem 4.8. ◀

Note that the schedulability test in Corollary 4.10 is only a sufficient schedulability test for the problem $1|period|_{\leq}D$. The only-if-part proof in Theorem 4.8 cannot be used for the problem $1|period|_{\leq}D$ because the task set may not permit such a feasible collection of jobs to be generated by **T**.

For implicit-deadline task systems, instead of testing all $t > 0$, it is possible to use the following sufficient schedulability test that can be validated in $O(n)$ time complexity.

▶ **Theorem 4.11.** *A set **T** of n implicit-deadline sporadic tasks is schedulable under EDF-NP for the problem $1|spor, impl|_{\leq}D$ if*

$$\begin{cases} \left(\sum_{i=1}^{k} U_i\right) + \frac{\max_{q=k+1}^{n} C_q}{T_k} \leq 1, \quad \forall k \in [n-1] \\ \sum_{i=1}^{n} U_i \leq 1 \end{cases} \tag{4.11}$$

where the n tasks are sorted such that $T_1 \leq T_2 \leq \cdots \leq T_n$.

Proof. This can be proved by approximating $\text{DBF}_i(t) \leq U_i t$ when $t \geq T_i$. The rest of the proof is left as an exercise. ◀

4.2.1 Time Complexity

Testing whether the schedulability condition of Eq. (4.8) in Theorem 4.8 in fact needs exponential time complexity. We will discuss the implementation of such a test in Section 4.5.

4.2.2 Optimality of EDF-NP

Although EDF-NP is not optimal for the problem $1|spor|_{\leq}D$, Jeffay et al. [30] show that EDF-NP is in fact optimal among non-preemptive work-conserving scheduling algorithms for implicit-deadline sporadic task sets, i.e., for the problem $1|spor, impl|_{\leq}D$. The proof by Jeffay et al. was based on a classical interchange argument. It is in fact not difficult to prove the optimality if the necessary condition in Theorem 4.8 is in fact also a necessary schedulability condition for any non-preemptive work-conserving scheduling algorithm.

▶ **Lemma 4.12.** *A set* $\boldsymbol{T}$ *of sporadic tasks in the problem* $1|spor|_{\leq}D$ *is schedulable under* ***any*** *non-preemptive* ***work-conserving*** *scheduling algorithm* ***only if***

$$\forall t > 0, \qquad \left(\sum_{\tau_i \in \boldsymbol{T}} \mathrm{DBF}_i(t)\right) + \max_{\tau_q : D_q > t} \{C_q - \epsilon\} \leq t \tag{4.12}$$

where $\epsilon > 0$ *is infinitesimal.*

Proof. The proof was already presented in the **only-if** proof of Theorem 4.8. Although the statement of Theorem 4.8 was for EDF-NP, the proof only used the property that EDF-NP is a work-conserving non-preemptive scheduler. Therefore, the necessary condition in Theorem 4.8 is actually for any work-conserving non-preemptive scheduling algorithm. ◀

As a result, we can reach the optimality of EDF-NP directly.

▶ **Theorem 4.13.** *Among non-preemptive work-conserving scheduling algorithms, EDF-NP is an optimal one for the problem* $1|spor|_{\leq}D$.

Proof. This is due to Theorem 4.8 and Lemma 4.12. ◀

Unfortunately, for periodic real-time task systems, EDF-NP is not an optimal non-preemptive work-conserving scheduling algorithm even when all tasks release their first jobs at the same time.[1]

▶ **Theorem 4.14.** *Among non-preemptive work-conserving scheduling algorithms, EDF-NP is not an optimal one for* $1|period, impl, O_i = 0|_{\leq}D$.

Proof. This can be proved by demonstrating a concrete task set that is not schedulable under EDF-NP but is schedulable under another work-conserving scheduling algorithm. The following example task set **T** with three tasks by Nasri and Fohler [44] demonstrates the non-optimality.

- $\tau_1 = \{C_1 = 1, T_1 = D_1 = 10, O_1 = 0\}$,

[1] The optimality statement in Theorem 5.1 by Jeffay et al. [30] is hence incorrect.

- $\tau_2 = \{C_2 = 8, T_2 = D_2 = 30, O_2 = 0\}$, and
- $\tau_3 = \{C_3 = 17, T_3 = D_3 = 60, O_3 = 0\}$.

To show that this task set is not schedulable under EDF-NP, we consider the feasible collection of jobs generated by **T**, in which $C_{i,j}$ is C_i for each task τ_i. Under EDF-NP, the second job $J_{1,2}$ of task τ_1 starts its execution at time $1+8+17 = 26$, which leads to a deadline miss, as its absolute deadline $d_{2,1}$ is 20.

The task set is in fact schedulable under a fixed-priority non-preemptive (FP-NP) scheduling algorithm which assigns $\tau_1 > \tau_3 > \tau_2$. The proof is left as an exercise. (Note that it is necessary to consider jobs whose actual execution times are shorter than their WCETs.) ◀

By Theorem 3.8, work-conserving scheduling algorithms are not optimal for the problem $1|r_j|_{\leq}D$. The same proof can be used to show that work-conserving scheduling algorithms are not optimal for $1|spor, cons|_{\leq}D$. How about the optimality of work-conserving scheduling algorithms for the simplest setting $1|period, impl, O_i = 0|_{\leq}D$? This class of scheduling algorithms is unfortunately also not optimal.

▶ **Theorem 4.15.** *Non-preemptive work-conserving scheduling algorithms are not optimal for the problem* $1|period, impl, O_i = 0|_{\leq}D$.

Proof. This can be proved by demonstrating a concrete task set that is not schedulable under any work-conserving scheduling algorithm, but can be in fact schedulable under a non-work-conserving scheduling algorithm. The following example task set **T** with three tasks demonstrates the non-optimality.

- $\tau_1 = \{C_1 = 1, T_1 = D_1 = 4, O_1 = 0\}$,
- $\tau_2 = \{C_2 = 1, T_2 = D_2 = 8, O_2 = 0\}$, and
- $\tau_3 = \{C_3 = 6, T_3 = D_3 = 16, O_3 = 0\}$.

To show that this task set is not schedulable under any non-preemptive work-conserving scheduling, we consider the feasible collection of jobs generated by **T**, in which $C_{i,j}$ is C_i for each task τ_i. Under any non-preemptive work-conserving scheduling, the first job $J_{3,1}$ of task τ_3 is either executed non-preemptively in time interval $(0,6]$, $(1,7]$, or $(2,8]$. In any of the three cases, one of the three jobs $J_{1,1}, J_{1,2}$ or $J_{2,1}$ misses its deadline.

Since the hyper-period is 16, we can use the same scehdule in the time interval $(0,16]$ repeatitively. This task set is in fact schedulable when $J_{1,1}$ is executed in $(0,1]$, $J_{2,1}$ is executed in $(1,2]$, $J_{1,2}$ is executed in $(4,5]$, $J_{3,1}$ is executed in $(6,11]$, $J_{1,3}$ is executed in $(11,12]$, $J_{1,4}$ is executed in $(12,13]$, and $J_{2,2}$ is executed in $(13,14]$. Figure 4.2 presents the above schedule. Note

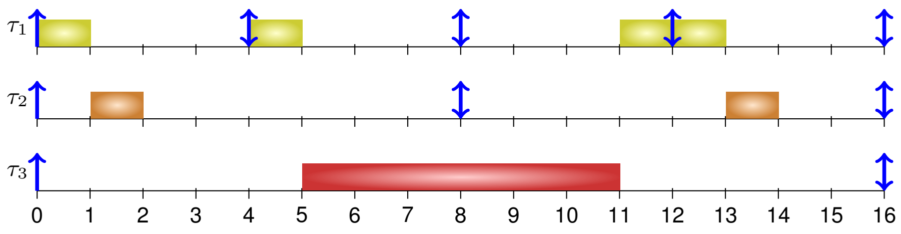


Figure 4.2: A feasible non-preemptive non-work-conserving schedule for the task set in the proof of Theorem 4.15.

that the above non-work-conserving schedule remains feasible for the task set **T** when the actual execution times of the jobs are shorter than their WCETs. ◀

4.2.3 Computational Complexity

Non-preemptiveness makes the scheduler design problem much harder. Even for the problem$1|period, O_i = 0, harmonic, impl|_{\leq}D$, finding an optimal schedule is $\mathcal{NP}$-hard in the strong sense, proved by Cai and Kong [10] due to a reduction from the well-known 3-PARTITION problem. This appealing industrial use case with harmonic periods has been studied by a series of researches [10, 21, 43]. More complicated periodic and sporadic task systems are in general more difficult to be handled. It should be clear now that an optimal non-preemptive schedule usually needs to insert some idle time; however there is no much advance with this respect. Extending EDF to non-work-conserving has been studied by Nasri and Fohler [44].

4.3 Fixed-Priority Preemptive Scheduling

As mentioned in Section 2.4, although EDF-P and EDF-NP scheduling algorithms provide nice properties for scheduling sporadic real-time tasks, such dynamic-priority scheduling algorithms are sometimes not available in real-time operating systems. Alternatively, fixed-priority (static-priority) scheduling algorithms are implemented in most real-time operating systems due to the simplicity of design and low management overhead.

Under preemptive fixed-priority (FP-P) scheduling, each task is assigned a unique priority before execution and does not change over time. The jobs generated by a task always have the same priority defined by the task. Here, we recap the notation of FP-P scheduling, as defined in Section 2.4 already: $hp(\tau_k)$ is the set of higher-priority tasks than task τ_k and $lp(\tau_k)$ is the set of lower-priority tasks than task τ_k. When task τ_i has a higher priority than task τ_j, we denote their priority relationship as $\tau_i > \tau_j$. We assume that the priority levels are unique.

4.3.1 Schedulability Tests of FP-P for Sporadic Task Systems

In Sections 4.1 and 4.2, we heavily use the notation $\mathbf{FJ}_{i,[r,r+t]}$ for analyzing EDF. For FP scheduling algorithms, we need another notation

$$\mathbf{FRJ}_{i,[r,r+\Delta)} = \{J_{i,j} \mid J_{i,j} \in \mathbf{FJ}_i, r_{i,j} \geq r, r_{i,j} < r + \Delta\} \tag{4.13}$$

That is, for a given feasible set $\mathbf{FJ}_i$ of jobs generated by a sporadic/periodic real-time task τ_i, let $\mathbf{FRJ}_{i,[r,r+\Delta)}$ be the subset of the jobs in $\mathbf{FJ}_i$ arriving in time interval $[r, r+\Delta)$.

▶ **Lemma 4.16.** *The total amount of execution time of the jobs of τ_i that are* ***released*** *in a time interval $[r, r+\Delta)$ for any $\Delta \geq 0$ is*

$$\sum_{J_{i,j} \in \boldsymbol{FRJ}_{i,[r,r+\Delta)}} C_{i,j} \leq \left\lceil \frac{\Delta}{T_i} \right\rceil C_i \overset{def}{=} demand_i(\Delta) \tag{4.14}$$

Proof. This comes from the definition of $\mathbf{FRJ}_{i,[r,r+\Delta)}$. Since $\mathbf{FRJ}_{i,[r,r+\Delta)}$ is also a feasible set of jobs generated by task τ_i, there are at most $\left\lceil \frac{\Delta}{T_i} \right\rceil$ jobs in $\mathbf{FRJ}_{i,[r,r+\Delta)}$, each with execution time no more than C_i. ◀

Lemma 4.16 is based on the demand *released* in a time interval. We can also define the demand *released and executed* in a time interval in a schedule. For this, we define the workload function $work_i(\Delta)$

$$work_i(\Delta) \overset{\text{def}}{=} \left\lfloor \frac{\Delta}{T_i} \right\rfloor C_i + \min\left\{C_i, \Delta - \left\lfloor \frac{\Delta}{T_i} \right\rfloor T_i\right\} \tag{4.15}$$

The workload function $work_i(\Delta)$ defined above is a piecewise function, i.e., linear in intervals $[\ell T_i, \ell T_i + C_i]$ with a slope 1 and constant, $(\ell+1)C_i$, in intervals $[\ell T_i + C_i, (\ell+1)T_i]$ for any non-negative integer ℓ.

▶ **Lemma 4.17.** *For a uniprocessor schedule σ and $\Delta \geq 0$,*

$$\int_r^{r+\Delta} \left(\mathbb{1}_{\sigma(t) \in \boldsymbol{FRJ}_{i,[r,r+\Delta)}}\right) dt \leq work_i(\Delta) \tag{4.16}$$

Proof. This comes from the definition of $\mathbf{FRJ}_{i,[r,r+\Delta)}$ together with a uniprocessor schedule σ. Since the schedule can only execute at most x amount of execution times in any time interval with length x, the function $work_i(t)$ defines the earliest processing of the jobs of task τ_i released in this interval $[r, r + \Delta)$. ◀

Figure 4.3 demonstrates an example of $demand_i(\Delta)$ and $work_i(\Delta)$. The following observations are based on the definitions of the functions $demand_i(\Delta)$ and $work_i(\Delta)$.

▶ **Observation 4.18.** *Functions $work_i(\Delta)$ and $demand_i(\Delta)$ are both non-decreasing with respect to Δ when $\Delta \geq 0$*

▶ **Observation 4.19.** *For any $\Delta - \delta \geq 0$,*

$$work_i(\Delta) \leq \delta + work_i(\Delta - \delta)$$

▶ **Observation 4.20.** *For any $\Delta \geq 0$,*

$$demand_i(\Delta) \geq work_i(\Delta)$$

Proof. This comes from the definition of the floor and ceiling functions. If Δ/T_i is a non-negative integer, then $demand_i(\Delta) = work_i(\Delta)$. If Δ/T_i is not an integer, then

$$work_i(\Delta) \leq \left\lfloor \frac{\Delta}{T_i} \right\rfloor C_i + C_i = \left\lceil \frac{\Delta}{T_i} \right\rceil C_i = demand_i(\Delta)$$

◀

4.3.1.1 Constrained-Deadline Task Systems

The above lemmas only quantify the demand of the jobs of task τ_i released no earlier than r. However, under fixed-priority scheduling, it is possible that a job of task τ_i is released before r and executed after r. Therefore, the following two lemmas are useful and can be applied to quantify the interference of the jobs of task τ_i that are released before r and executed after r. Lemma 4.21 is less precise based on the demand function from Lemma 4.16, but is used widely in the literature. Lemma 4.22 is applicable if $R_i \leq T_i$ and more precise, but the analysis is more complicated, presented by Chen et al. [14].

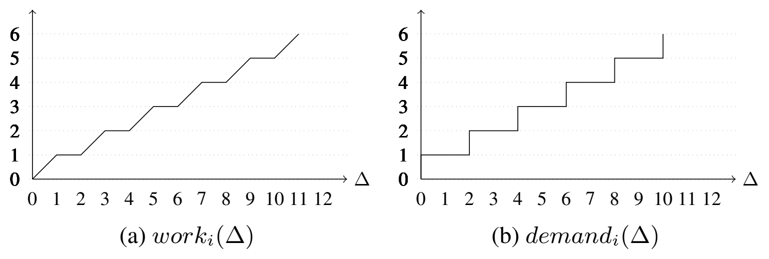


Figure 4.3: An example of the functions $work_i(\Delta)$ and $demand_i(\Delta)$: $\tau_i = \{T_i = D_i = 2, C_i = 1\}$ in this example.

▶ **Lemma 4.21.** *Suppose that the worst-case response time of a sporadic/periodic task τ_i is no more than R_i, where $R_i \geq C_i$, in a schedule σ. The amount of execution times of the jobs generated by τ_i **executed** in a time interval $(r, r+\Delta]$ for any $\Delta \geq 0$ in the schedule σ is no more than $demand_i(\Delta+R_i)$. That is,*

$$\int_r^{r+\Delta} \mathbb{1}_{\sigma(t)\in \boldsymbol{FJ}_i} \leq work_i(\Delta + R_i)$$

Proof. By the definition of the worst-case response time, the jobs of task τ_i executed in a time interval $(r, r + \Delta]$ cannot be released before $r - R_i$. By Lemma 4.16, the total amount of execution time of the jobs of τ_i that are *released* in a time interval $[r - R_i, r + \Delta)$ for any $\Delta \geq 0$ is $demand_i(\Delta + R_i)$. Therefore, the amount of time executed for task τ_i in the time interval $(r + r + \Delta]$ is no more than $demand_i(\Delta + R_i)$. ◀

▶ **Lemma 4.22.** *Suppose that the worst-case response time of a sporadic/periodic task τ_i is no more than R_i, where $C_i \leq R_i \leq T_i$, in a schedule σ. The amount of execution times of the jobs generated by τ_i **executed** in a time interval $(r, r + \Delta]$ for any $\Delta \geq 0$ in the schedule σ is no more than* $\min\{\Delta, work_i(\Delta + R_i - C_i)\}$. *That is,*

$$\int_r^{r+\Delta} \mathbb{1}_{\sigma(t)\in \boldsymbol{FJ}_i} \leq \min\{\Delta, work_i(\Delta + R_i - C_i)\}$$

Proof. By the definition of the worst-case response time, the jobs of task τ_i executed in a time interval $(r, r + \Delta]$ cannot be released before $r - R_i$. Since $R_i \leq T_i$, there is at most one job, denoted as $J_{i,j}$, of task τ_i released before

r and executed after r. Suppose that c_i^* is the amount of time $J_{i,j}$ is executed in $(r, r+\Delta]$, i.e., $c_i^* = \int_r^{r+\Delta} \mathbb{1}_{\sigma(t)=J_{i,j}} dt$. By definition $0 \le c_i^* \le C_i$.

When c_i^* is 0, we can directly apply Lemma 4.17 and Observation 4.18 to conclude this case. Suppose that $c_i^* > 0$ for the rest of the proof. In schedule σ, let ρ_i be the time when $J_{i,j}$ finishes its execution. By the definition of ρ_i,

$$\rho_i - r \ge c_i^*$$

By the definition of the worst-case response time and the above inequality, the arrival time $r_{i,j}$ of job $J_{i,j}$ must be

$$r_{i,j} \ge \rho_i - R_i \ge c_i^* + r - R_i$$

For a sporadic (periodic, respectively) task τ_i, the next job of task τ_i is released no earlier than (at, respectively)

$$r_{i,j} + T_i \ge c_i^* + r - R_i + T_i$$

The case when $c_i^* + r - R_i + T_i \ge r + \Delta$ is obvious since there is only one job of task τ_i executed in this scenario, i.e., the workload executed in $(r, r+\Delta]$ is at most $work_i(\Delta) \le work_i(\Delta + R_i - C_i)$. We focus on another case $c_i^* + r - R_i + T_i < r + \Delta$. Since the arrival time of the first job of task τ_i released no earlier than $r + \rho_i$ is at least $c_i^* + r - R_i + T_i$, we know that $\mathbf{FJ}_{i,[r+\rho_i, r+\Delta)}$ is the same as $\mathbf{FJ}_{i,[c_i^*+r-R_i+T_i, r+\Delta)}$, and

$$\int_{r+\rho_i}^{r+\Delta} \mathbb{1}_{\sigma(t)\in \mathbf{FJ}_{i,[r+\rho_i,r+\Delta)}} dt = \int_{c_i^*+r-R_i+T_i}^{r+\Delta} \mathbb{1}_{\sigma(t)\in \mathbf{FJ}_{i,[c_i^*+r-R_i+T_i,r+\Delta)}}$$

Therefore,

$$\begin{aligned}
\int_r^{r+\Delta} \mathbb{1}_{\sigma(t)\in \mathbf{FJ}_i} dt &= \int_r^{r+\rho_i} \mathbb{1}_{\sigma(t)=J_{i,j}} dt + \int_{r+\rho_i}^{r+\Delta} \mathbb{1}_{\sigma(t)\in \mathbf{FJ}_i} dt \\
&= \int_r^{r+\rho_i} \mathbb{1}_{\sigma(t)=J_{i,j}} dt + \int_{r+\rho_i}^{r+\Delta} \mathbb{1}_{\sigma(t)\in \mathbf{FJ}_{i,[r+\rho_i,r+\Delta)}} dt \\
&\le c_i^* + \int_{r+\rho_i}^{r+\Delta} \mathbb{1}_{\sigma(t)\in \mathbf{FJ}_{i,[r+\rho_i,r+\Delta)}} dt \\
&= c_i^* + \int_{c_i^*+r-R_i+T_i}^{r+\Delta} \mathbb{1}_{\sigma(t)\in \mathbf{FJ}_{i,[c_i^*+r-R_i+T_i,r+\Delta)}} dt \\
&\le c_i^* + work_i(\Delta + R_i - c_i^* - T_i)
\end{aligned}$$

where the last inequality is due to Lemma 4.17 since $\Delta + R_i - c_i^* - T_i > 0$ in our assumption.

For any $0 < c_i^* \leq C_i$, we can now prove that $work_i(\Delta + R_i - C_i) \geq c_i^* + work_i(\Delta + R_i - c_i^* - T_i)$. Figure 4.4 provides an illustrative example. We consider three subcases:

- For $0 \leq \Delta \leq C_i$, since $R_i - T_i \leq 0$, by Observation 4.18,
$$\begin{aligned} c_i^* + work_i(\Delta + R_i - c_i^* - T_i) \leq & c_i^* + work_i(\Delta - c_i^*) \\ = & \Delta = work_i(\Delta) \leq work_i(\Delta + R_i - C_i) \end{aligned}$$
where the last inequality is due to the fact $R_i - C_i \geq 0$.
- For $C_i < \Delta \leq T_i - R_i + C_i$, by Observation 4.18,
$$\begin{aligned} & c_i^* + work_i(\Delta + R_i - c_i^* - T_i) \\ \leq & c_i^* + work_i(C_i - c_i^*) = C_i = work_i(C_i) \leq work_i(\Delta + R_i - C_i) \end{aligned}$$
where the last inequality is due to the fact $R_i - C_i \geq 0$ and the assumption $\Delta > C_i$.
- For $T_i - R_i + C_i < \Delta$, let δ be $C_i - c_i^*$. Then, by Observation 4.19,
$$\begin{aligned} & c_i^* + work_i(\Delta + R_i - c_i^* - T_i) \\ \leq & c_i^* + \delta + work_i(\Delta + R_i - c_i^* - T_i - \delta) \\ = & C_i + work_i(\Delta + R_i - C_i - T_i) \\ = & work_i(\Delta + R_i - C_i - T_i + T_i) \\ = & work_i(\Delta + R_i - C_i) \end{aligned}$$

Therefore, we reach the conclusion. ◀

With the above lemmas to quantify the maximum amount of time task τ_i can be executed in an interval $(r, r + \Delta]$, we can now present a safe worst-case response time analysis (schedulability test) for fixed-priority preemptive uniprocessor scheduling algorithms.

▶ **Theorem 4.23.** *Suppose that the worst-case response time of task τ_i is at most R_i, in which $C_i \leq R_i$, for every higher-priority task $\tau_i \in hp(\tau_k)$. Let $\Delta_{\min} > 0$ be the minimum value that satisfies*

$$\Delta_{\min} = C_k + \sum_{\tau_i \in hp(\tau_k)} work_i(R_i + \Delta_{\min}) \tag{4.17}$$

The WCRT R_k of task τ_k in the problem $1|spor, prmp, fp|RT$ is

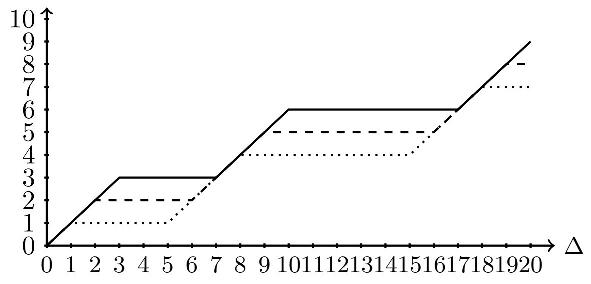


Figure 4.4: An example of when $T_i = 10$, $C_i = 3$, and $R_i = 6$ in the proof of Lemma 4.22. Solid line: c_i^* is 3, Dashed line: c_i^* is 2, Dotted line: c_i^* is 1.

- $R_k \leq \Delta_{\min}$, *if* $\Delta_{\min} \leq T_k$.

Note that when $\Delta_{\min} > T_k$*, the WCRT of task* τ_k *can not be determined by this method.*

Proof. There exists a feasible collection of jobs generated by $\mathbf{T}$ where task τ_k reaches its worst-case response time R_k. Let $\mathbf{FJ}$ be such a collection of jobs, where $\mathbf{FJ}_i$ is its subset generated by a sporadic real-time task τ_i in $\mathbf{T}$ for every task $\tau_i \in \mathbf{T}$. Let $\sigma : \mathbb{R} \rightarrow \mathbf{FJ} \cup \{\perp\}$ be the FP-P schedule for $\mathbf{FJ}$.

In schedule σ, we consider two cases. **Case 1:** $R_k \leq T_k$. Then, let $J_{k,\ell}$ be a job of task τ_k which has the worst-case response time R_k. Therefore, for any $0 < \Delta < R_k$,

$$\int_{r_{k,\ell}}^{r_{k,\ell}+\Delta} \mathbb{1}_{\sigma(t)=J_{k,\ell}} dt < C_{k,\ell} \leq C_k \tag{4.18}$$

Due to fixed-priority preemptive scheduling, during $r_{k,\ell}$ and $r_{k,\ell} + R_k$ only $J_{k,\ell}$ or higher-priority jobs generated by $hp(\tau_k)$ are executed in σ. Therefore, for any $0 < \Delta < R_k$,

$$\begin{aligned}
\Delta = \quad & \int_{r_{k,\ell}}^{r_{k,\ell}+\Delta} \mathbb{1}_{\sigma(t)\in J_{k,\ell}} dt + \sum_{\tau_i \in hp(\tau_k)} \int_{r_{k,\ell}}^{r_{k,\ell}+\Delta} \mathbb{1}_{\sigma(t)\in \mathbf{FJ}_i} dt \\
\overset{\text{Eq. (4.18)}}{<} \quad & C_k + \sum_{\tau_i \in hp(\tau_k)} \int_{r_{k,\ell}}^{r_{k,\ell}+\Delta} \mathbb{1}_{\sigma(t)\in \mathbf{FJ}_i} dt \\
\overset{\text{Lemma 4.21}}{\leq} \quad & C_k + \sum_{\tau_i \in hp(\tau_k)} work_i(R_i + \Delta)
\end{aligned}$$

Case 2: $R_k > T_k$. Then, let $J_{k,\ell}$ be the first job of task τ_k which has not finished its execution before $r_{k,\ell} + T_{k,\ell}$. Therefore, the analysis in the first case remains valid for any $0 < \Delta \leq T_k$.

With the above two cases, the minimum positive Δ such that $\Delta = C_k + \sum_{\tau_i \in hp(\tau_k)} work_i(R_i + \Delta)$ is a safe upper bound on R_k if it is no more than T_k. ◀

▶ **Corollary 4.24.** *Suppose that the worst-case response time of task τ_i is at most R_i, in which $C_i \leq R_i \leq T_i$, for every higher-priority task $\tau_i \in hp(\tau_k)$. Let $\Delta_{\min} > 0$ be the minimum value that satisfies*

$$\Delta_{\min} = C_k + \sum_{\tau_i \in hp(\tau_k)} work_i(R_i - C_i + \Delta_{\min}) \tag{4.19}$$

The WCRT R_k of task τ_k in the problem $1|spor, prmp, fp|RT$ is

- *$R_k \leq \Delta_{\min}$ if $\Delta_{\min} \leq T_k$.*

Note that when $\Delta_{\min} > T_k$, the WCRT of task τ_k can not be determined by this method.

Proof. This follows the same proof procedure by applying Lemma 4.22 instead of Lemma 4.21 during the proof. ◀

The analysis in Theorem 4.23 and Corollary 4.24 only considers the schedule σ from a job of task τ_k. The interval of interest always starts from the arrival time $r_{k,\ell}$ of the job of task τ_k under analysis in both proofs. This type of analysis is safe but can be further improved.

One key observation is that *fixed-priority preemptive scheduling* is work-conserving. Suppose that the schedule σ is busy executing jobs with higher priorities than job $J_{k,\ell}$ in the time interval $[t, r_{k,\ell}]$. It is possible to extend the interval of interest from $r_{k,\ell}$ towards left until the processor idles or executes a job released from $lp(\tau_k) \cup \{\tau_k\}$. This extension results in an exact schedulability test.

▶ **Theorem 4.25.** *Let $\Delta_{\min} > 0$ be the minimum value that satisfies*

$$\Delta_{\min} = C_k + \sum_{\tau_i \in hp(\tau_k)} demand_i(\Delta_{\min}) \tag{4.20}$$

The WCRT R_k of task τ_k in the problem $1|spor, prmp, fp|RT$ is

- *$R_k = \Delta_{\min}$, if $\Delta_{\min} \leq T_k$, and*

- *$R_k > T_k$, otherwise.*

*Note that if $\Delta_{\min} \leq T_k$, the evaluated worst-case response time R_k is **exact**, independent from the deadline satisfactions of the higher-priority tasks.*

Proof. To prove this theorem, we revise the proof of Theorem 4.23 in **Case 1:** $R_k \leq T_k$. Then, let $J_{k,\ell}$ be a job of task τ_k which has the worst-case response time R_k. Let t_0 be the earliest instant prior to $r_{k,\ell}$, i.e., $t_0 \leq r_{k,\ell}$, such that the processor only executes jobs generated by the higher-priority tasks in $hp(\tau_k)$ in time interval $(t_0, r_{k,\ell}]$ under FP-P. That means, immediately prior to time t_0, i.e., $t = t_0 - \epsilon$ for an infinitesimal ϵ, $\sigma(t)$ is either $\perp$ or a job of $\{\tau_k\} \cup lp(\tau_k)$. We note that t_0 exists.

With the above definition, $\sum_{\tau_i \in hp(\tau_k)} \int_{t_0}^{r_{k,\ell}} \mathbb{1}_{\sigma(t)\in \mathbf{FJ}_i} dt = r_{k,\ell} - t_0$ and $\int_{t_0}^{r_{k,\ell}} \mathbb{1}_{\sigma(t)\in \mathbf{FJ}_k} dt = 0$. Therefore, for any $0 < \Delta < R_k + r_{k,\ell} - t_0$,

$$\int_{t_0}^{t_0+\Delta} \mathbb{1}_{\sigma(t)=J_{k,\ell}} dt < C_{k,\ell} \leq C_k \tag{4.21}$$

Due to fixed-priority preemptive scheduling, during t_0 and $r_{k,\ell} + R_k$ only $J_{k,\ell}$ or higher-priority jobs generated by $hp(\tau_k)$ **released no earlier than** t_0, i.e., jobs in $\cup_{\tau_i \in hp(\tau_k)} \mathbf{FRJ}_{i,[t_0,t_0+\Delta)}$, are executed in σ. Therefore, for any $0 < \Delta < R_k + r_{k,\ell} - t_0$,

$$\begin{aligned}
\Delta = \quad & \int_{t_0}^{t_0+\Delta} \mathbb{1}_{\sigma(t)\in J_{k,\ell}} dt + \sum_{\tau_i \in hp(\tau_k)} \int_{t_0}^{t_0+\Delta} \mathbb{1}_{\sigma(t)\in \mathbf{FRJ}_{i,[t_0,t_0+\Delta)}} dt \\
\overset{\text{Eq. (4.21)}}{<} \quad & C_k + \sum_{\tau_i \in hp(\tau_k)} \int_{t_0}^{t_0+\Delta} \mathbb{1}_{\sigma(t)\in \mathbf{FRJ}_{i,[t_0,t_0+\Delta)}} dt \\
\overset{\text{Lemma 4.16}}{\leq} \quad & C_k + \sum_{\tau_i \in hp(\tau_k)} demand_i(\Delta)
\end{aligned}$$

Hence, the minimum positive Δ such that $\Delta = C_k + \sum_{\tau_i \in hp(\tau_k)} demand_i(\Delta)$ is a safe upper bound on R_k if it is no more than T_k. This value of $\Delta_{\min}$ is in fact **exact** for sporadic real-time task systems. To prove this, we simply release the first jobs of $\mathbf{T}$ at time 0, and the subsequent jobs are released as early as possible by respecting the minimum inter-arrival time. Therefore, the jobs of task τ_i are released at time $0, T_i, 2T_i, \ldots$, etc. All jobs use their corresponding worst-case execution times. Due to the work-conserving execution property of the FP-P, the first job of task τ_k only finishes at time $\Delta_{\min}$ since for $0 < \Delta < \Delta_{\min}$

$$C_k + \sum_{\tau_i \in hp(\tau_k)} \int_0^{\Delta} \mathbb{1}_{\sigma(t) \in \mathbf{FJ}_i} dt > \Delta$$

As for **Case 2:** $R_k > T_k$. Then, let $J_{k,\ell}$ be the first job of task τ_k which has not finished its execution before $r_{k,\ell} + T_{k,\ell}$. Therefore, the analysis in the first case remains valid for any $0 < \Delta \leq T_k$.

◀

Theorem 4.25 can be re-written into a more popular form, called time-demand analysis (TDA) proposed by Lehoczky et al. [**?**]: A constrained-deadline task τ_k is schedulable under FP-P scheduling if and only if

$$\exists t | 0 < t \leq D_k, \quad C_k + \sum_{\tau_i \in hp(\tau_k)} \left\lceil \frac{t}{T_i} \right\rceil C_i \leq t \tag{4.22}$$

TDA might seem difficult as it requires to check every time t with $0 < t \leq D_k$ for a given τ_k. There are two ways to avoid this:

- Iterate using $t(\ell + 1) := W_k(t(\ell))$, starting with $t(0) := \sum_{j=1}^{k} C_j$ and stopping, when $t(\ell) = W_k(t(\ell))$ or $t(\ell) > D_i$ for some ℓ.
- Only consider $t \in \{\ell T_j \mid 1 \leq j \leq i, \ell \in \mathcal{N}^+\}$. That is, only consider t at which a job of higher-priority tasks arrives.

4.3.1.2 Critical Instant Theorem and Historical Perspectives

Theorem 4.25 is a very interesting and remarkable result, widely used in the literature. It suggests to validate the worst-case response time of task τ_k by

- releasing the first jobs of the higher-priority tasks in $hp(\tau_k)$ together with a job of τ_k and
- releasing the subsequent jobs of the higher-priority tasks in $hp(\tau_k)$ as early as possible by respecting their minimum inter-arrival times.

To explain the above phenomena, Liu and Layland in their seminal paper [40] in 1973 defined two terms (according to their wording):

- A **critical instant** for task τ_k is an instant at which a job of task τ_k released at this instant has the largest response time.
- A **critical time zone** for task τ_k is a time interval starting from a critical instant of τ_k to the completion of the job of task τ_k released at the critical instant.

Liu and Layland [40] concluded the famous **critical instant theorem** as follows: "*A critical instant for any task occurs whenever the task is requested simultaneously with requests for all higher priority tasks*." Their proof was in fact incomplete because they did not consider to extend the interval of interest towards t_0 like the proof of Theorem 4.25. Moreover, their definition of the critical instant theorem was incomplete since the condition $\Delta_{\min} > T_k$ was not considered in their definition. A **precise definition of the critical instant theorem** is revised as follows:

- A **critical instant** for task τ_k is an instant such that
  - a job of task τ_k released at this instant has the largest response time if it is no more than T_k, or
  - the worst-case response time of a job of task τ_k released at this instant is more than T_k.
- A **critical time zone** for task τ_k is a time interval starting from a critical instant of τ_k to the completion of the job of task τ_k released at the critical instant.
- In a critical time zone for task τ_k, all the tasks release their first jobs at a critical instant for task τ_k and their subsequent jobs as early as possible by respecting their minimum inter-arrival times.

4.3.1.3 Arbitrary-Deadline Task Systems

When defining t_0 in the proofs of Theorems 4.23 and 4.25, we assume that there is only one job $J_{k,\ell}$ of task τ_k in the analysis window $(t_0, d_{k,\ell}]$. This assumption is correct when $R_k \leq T_k$ but incorrect when $R_k > T_k$. When there is at least one unfinished job of task τ_k (arrived before $r_{k,\ell}$), the proofs of Theorems 4.23 and 4.25 cannot be applied correctly.

To handle the jobs of task τ_k carried into the window of analysis, the concept of *level-k busy interval* has been widely used in the literature. Informally, the level-k busy interval is the longest interval such that there is at least a job of task τ_k unfinished before a job of task τ_k arrives in this interval. One intuitive idea is to extend the critical time zone of task τ_k by continuing the release of the jobs of task τ_k *periodically* until the moment that there is no job of task τ_k unfinished yet.

Consider an example with two tasks: τ_1 has $T_1 = 70$ and $C_1 = 26$ and τ_2 is with $T_2 = 100$ and $C_2 = 62$. Figure 4.5 illustrates the schedule, assuming the critical time zone. The response times of the 7 jobs of τ_2 illustrated in Figure 4.5 are 114, 102, 116, 104, 118, 106, and 94. Note that the first job's response time of τ_2 is not the longest.

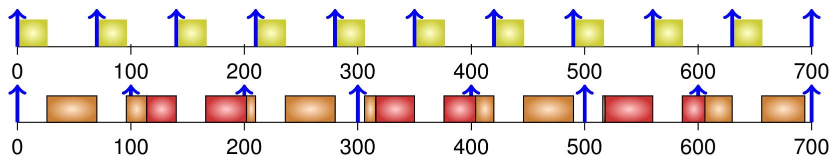


Figure 4.5: Arbitrary-deadline tasks under fixed-priority scheduling

In the following theorem, we show that Eq. (4.22) defines the length of the level-k busy interval and Eq. (4.23) defines the worst-case response time of task τ_k. Specifically, Eq. (4.22) supports the correctness of the extension of the critical time zone mentioned above.

▶ **Theorem 4.26.** *For any positive integer h, let $\Delta_{\min,h} > 0$ be the minimum value that satisfies*

$$\Delta_{\min,h} = hC_k + \sum_{\tau_i \in hp(\tau_k)} demand_i(\Delta_{\min,h}) \tag{4.23}$$

Let h^* be the minimum (positive) integer h such that $\Delta_{\min,h} \leq hT_k$. The WCRT R_k of task τ_k in the problem $1|spor, prmp, fp|RT$ is

$$R_k = \max_{h \in [h^*]} \{\Delta_{\min,h} - (h-1)T_k\} \tag{4.24}$$

*Note that the evaluated worst-case response time R_k is **exact**.*

Proof. **Case 1:** $R_k \leq T_k$ We have already reached the result from Theorem 4.25. In such a case, h^* is 1 by definition.

We consider **Case 2:** $R_k > T_k$ in the proof of Theorem 4.23. Let $J_{k,\ell}$ be the job of task τ_k which has the worst-case response time. Let t_0' be the earliest instant prior to $r_{k,\ell}$, i.e., $t_0' \leq r_{k,\ell}$, such that the processor only executes jobs generated by either task τ_k or the higher-priority tasks in $hp(\tau_k)$ in time interval $(t_0', r_{k,\ell}]$ under FP-P. That means, immediately prior to time t_0', i.e., $t = t_0' - \epsilon$ for an infinitesimal ϵ, $\sigma(t)$ is either $\perp$ or a job of $lp(\tau_k)$. We note that t_0' exists.

Between t_0' and $r_{k,\ell}$, if there exists any instant at which a job of task τ_k finishes but there is no other job of task τ_k released beforehand, let t_0 be the last one. Otherwise, let t_0 be t_0'. We note that t_0 exists. Suppose that there are ℓ jobs of task τ_k released in time interval $[t_0, r_{k,\ell}]$. For notational brevity, let these jobs be indexed as $J_{k,1}, J_{k,2}, \ldots, J_{k,\ell}$. With the above definition, $\ell \geq 2$ and

$$\int_{r_{k,g}}^{r_{k,g}+\Delta} \mathbb{1}_{\sigma(t)=J_{k,g}} dt < C_{k,g} \leq C_k, \ \forall g \in [\ell-1], 0 < \Delta < r_{k,g+1} - r_{k,g} \tag{4.25}$$

$$\int_{r_{k,\ell}}^{r_{k,\ell}+\Delta} \mathbb{1}_{\sigma(t)=J_{k,\ell}} dt < C_{k,\ell} \leq C_k, \ \forall 0 < \Delta < R_k \tag{4.26}$$

The inequality in Eq. (4.24) holds because at least one job of task τ_k is not finished yet before a job of task τ_k arrived in time interval $[t_0, r_{k,\ell}]$. The inequality in Eq. (4.25) holds because R_k is the worst-case response time of τ_k.

Due to fixed-priority preemptive scheduling, during t_0 and $r_{k,2}$ only $J_{k,1}$ or higher-priority jobs generated by $hp(\tau_k)$ **released no earlier than** t_0, i.e., jobs in $\cup_{\tau_i \in hp(\tau_k)} \textbf{FRJ}_{i,[t_0,t_0+\Delta)}$, are executed in σ. Therefore, for any $0 < \Delta < r_{k,g+1} - t_0$, $\forall g \in [\ell - 1]$,

$$\begin{aligned}
\Delta = & \sum_{j=1}^{g} \int_{t_0}^{t_0+\Delta} \mathbb{1}_{\sigma(t)\in J_{k,j}} dt + \sum_{\tau_i \in hp(\tau_k)} \int_{t_0}^{t_0+\Delta} \mathbb{1}_{\sigma(t)\in \textbf{FRJ}_{i,[t_0,t_0+\Delta)}} dt \\
\overset{\text{Eq. (4.24)}}{<} & \ gC_k + \sum_{\tau_i \in hp(\tau_k)} \int_{t_0}^{t_0+\Delta} \mathbb{1}_{\sigma(t)\in \textbf{FRJ}_{i,[t_0,t_0+\Delta)}} dt \\
\overset{\text{Lemma 4.16}}{\leq} & \ gC_k + \sum_{\tau_i \in hp(\tau_k)} demand_i(\Delta)
\end{aligned}$$

Therefore, $\forall g \in [\ell - 1]$

$$\Delta_{\min,g} > r_{k,g+1} - t_0 = r_{k,g+1} - r_{k,1} + r_{k,1} - t_0 \geq gT_i$$

Similarly, for $0 < \Delta < r_{k,\ell} + R_k$

$$\Delta < \ell C_k + \sum_{\tau_i \in hp(\tau_k)} demand_i(\Delta)$$

And,

$$\Delta_{\min,\ell} \geq r_{k,\ell} + R_k - t_0 = R_k + r_{k,\ell} - r_{k,1} + r_{k,1} - t_0 \geq R_k + (\ell-1)T_i$$

Note that since the jobs of task τ_k are executed in the FCFS manner, among the jobs of task τ_k, only the jobs in $\mathbf{FRJ}_{k,[r_{k,1},r_{k,\ell}]}$ are executed before job $J_{k,\ell}$ finishes. Therefore, the finishing time of job $J_{k,\ell}$ is at most $t_0 + \Delta_{\min,\ell}$. Therefore,

$$R_k \leq \Delta_{\min,\ell} - (\ell - 1)T_i$$

Therefore, the theorem holds for a safe upper bound on the worst-case response time. The calculated R_k is in fact **exact** for sporadic real-time task systems. To prove this, we simply release the first jobs of $\mathbf{T}$ at time 0, and the subsequent jobs are released as early as possible by respecting the minimum inter-arrival time. Therefore, the jobs of task τ_i are released at time $0, T_i, 2T_i, \ldots$, etc. All jobs use their corresponding worst-case execution times. It can be observed that the job $J_{k,h}$ arrives exactly at $(h-1)T_k$ and finishes exactly at time $\Delta_{\min,h}$ for any $h \in [h^*]$. ◀

4.3.2 Schedulability Tests of FP-P for Periodic Task Systems

After discussing the schedulability test problem $1|spor, fp, prmp|_{\leq}D$, we now focus on its special case $1|period, fp, prmp|_{\leq}D$. Since a periodic task set is a special case of a sporadic task set, the schedulability tests in Theorems 4.25 and 4.26 can be applied safely as sufficient schedulability tests.

In other words, the critical instant theorem (or critical time zone) and the level-k busy interval are sufficient conditions for periodic task systems. But, are they also necessary conditions for periodic task systems?

The same example in Example 4.5 can be used to demonstrate the pessimism of the test based on Theorem 4.25. Suppose that τ_1 has a higher priority than τ_2. We immediately conclude that τ_2 misses its deadline since its worst-case response time (under the critical time zone) is 4. However, this task set is in fact schedulable under any work-conserving scheduling algorithm since the jobs released by the two tasks are perfectly separate from each other without any execution interference.

For a given set $\mathbf{T}$ of periodic real-time tasks, the offset O_i of every task τ_i is known in advance. To evaluate the worst-case response time of task τ_k, we simply have to simulate the schedule. However, since jobs may finish earlier than their worst-case execution time, it may be necessary to simulate all possible execution scenarios. Fortunately, under uniprocessor fixed-priority preemptive scheduling, it is not difficult to prove that the worst-case response time of τ_k can be exactly derived by simulating a schedule of feasible collections of jobs in which every job runs its worst-case execution time.

The remaining question is the length of the simulated schedule. Suppose that $O_{\max} = \max_{\tau_i \in \mathbf{T}} O_i$ and H is the hyper-period of $\mathbf{T}$. For the case $1|period, fp, cons, prmp|_{\leq}D$, Leung and Whitehead [37] showed that if there is a deadline miss, such a deadline miss can be observed in time interval $(0, O_{\max} + 2H]$, provided that $\sum_{\tau_i \in \mathbf{T}} U_i \leq 1$. Therefore, if $\sum_{\tau_i \in \mathbf{T}} U_i \leq 1$ and there is no deadline miss at all in the resulting schedule σ in time interval $(0, O_{\max} + 2H]$, then there is no deadline miss for the problem $1|period, fp, cons, prmp|_{\leq}D$. *This test is necessary and sufficient.*

4.3.3 Optimality of RM-P and DM-P

The rate-monotonic (RM) scheduling algorithm uses a simple rule: priorities are assigned to tasks according to their request rates. That is, tasks with higher request rates (i.e., shorter periods) have higher priorities, in which ties are broken arbitrarily. In 1973, Liu and Layland [40] showed that preemptive RM (RM-P) is optimal among all fixed-priority assignments under preemptive fixed-priority scheduling algorithms for periodic task systems.

Note that Liu and Layland in fact over-stated the result in their paper [40] in 1973. RM-P is only optimal for $1|period, O_i = 0, impl, fp, prmp|_{\leq}D$ or $1|spor, impl, fp, prmp|_{\leq}D$, but it is not optimal if the periodic tasks have different offsets, i.e., $1|period, impl, fp, prmp|_{\leq}D$. We will take a deeper look into the gap in this section.

In 1982, Leung and Whitehead [37] further considered constrained-deadline task systems and proposed preemptive deadline-monotonic (DM-P) scheduling algorithm: priorities are assigned to tasks according to their relative deadlines. DM-P was then also extended to arbitrary-deadline task systems. In 1990, Lehoczky [35] showed that DM-P is not optimal for the problem $1|spor, arb, fp, prmp|_{\leq}D$. Note that DM-P is identical to RM-P if the task set $\mathbf{T}$ is an implicit-deadline task set.

▶ **Theorem 4.27.** *The rate-monotonic priority assignment is optimal for the problem* $1|period, O_i = 0, impl, fp, prmp|_{\leq}D$*. The deadline-monotonic priority assignment is optimal for* $1|period, O_i = 0, cons, fp, prmp|_{\leq}D$*.*

▶ **Theorem 4.28.** *The deadline-monotonic priority assignment is optimal for* $1|spor, cons, fp, prmp|_{\leq}D$*.*

Proof. This is left as an exercise in Exercise 4.5. ◀

Although Theorem 4.27 shows the optimality of RM-P and DM-P when the periodic tasks have the same offset accordingly, the optimality does not hold for periodic tasks with asynchronous offsets. If the periodic tasks do not have the same phase (offset), Leung and Whitehead [37] showed RM-P is not the optimal fixed-priority preemptive priority assignment *in the weak sense*. That is, there exists a priority assignment under the rate-monotonic policy, which is not optimal for the problem $1|period, impl, fp, prmp|_{\leq}D$. This can be showen by a concrete task set **T** (by Leung and Whitehead [37]):

- $\tau_1 = \{C_1 = 3, T_1 = D_1 = 8, O_1 = 0\}$,
- $\tau_2 = \{C_2 = 1, T_2 = D_2 = 12, O_2 = 10\}$, and
- $\tau_3 = \{C_3 = 6, T_3 = D_3 = 12, O_3 = 0\}$.

For this task set, task τ_2 and task τ_3 have the same period. Therefore, there are two different rate-monotonic priority assignments. One assigns task τ_2 as the lowest-priority task and another assigns task τ_3 as the lowest-priority task. In both cases, task τ_1 has the highest priority under RM-P. It is not difficult to show (left as an exercise) that assigning τ_3 as the lowest-priority task results in a deadline miss of task τ_3, whilst assigning τ_2 as the lowest-priority task is a feasible fixed-priority schedule for the given task set **T**.

Although the above example shows that not all rate-monotonic priority assignments are optimal for the problem $1|period, impl, fp, prmp|_{\leq}D$, it still does not exclude the possibility that one of the RM-P priority assignments is optimal. Goossens [24] presented the following example, showing that the unique RM-P priority assignment of the following task set **T** is not an optimal one for the problem $1|period, impl, fp, prmp|_{\leq}D$:

- $\tau_1 = \{T_1 = D_1 = 10, C_1 = 7, O_1 = 0\}$,
- $\tau_2 = \{T_2 = D_2 = 15, C_2 = 3, O_2 = 4\}$, and
- $\tau_3 = \{T_3 = D_3 = 16, C_3 = 1, O_3 = 0\}$.

Under RM-P (i.e., $\tau_1 > \tau_2 > \tau_3$), the first job of task τ_3 misses its deadline since $2 \times 7 + 3 + 1 = 18 > 16$. The task set is in fact schedulable under another priority assignment $\tau_1 > \tau_3 > \tau_2$, which can be shown by simulating the (worst-case) schedule from time 0 to 244.

▶ **Theorem 4.29.** *The rate-monotonic priority assignment is not optimal for the problem* $1|period, impl, fp, prmp|_{\leq}D$.

Proof. This is due to the task set discussed above. ◀

4.3.4 Optimal Priority Assignment (OPA)

4.4 Fixed-Priority Non-Preemptive Scheduling

This section considers non-preemptive uniprocessor scheduling algorithms for sporadic real-time task systems under FP-NP scheduling algorithm. Under FP-NP, the job in the ready queue whose (task) priority is the highest is executed *non-preemptively* on the processor.

4.4.1 Schedulability Tests of FP-NP

Under non-preemptive fixed-priority scheduling, a lower-priority job that has started its execution can block a higher-priority job that arrives later. Fortunately, such lower-priority blocking happens only once after a job of τ_k (under analysis) is ready. What we need to do is to consider the longest execution time $(\max_{\tau_i \in lp(\tau_k)} C_i)$ of the lower-priority jobs in $lp(\tau_k)$ as the *blocking time*. The following theorem provides a sufficient test.

▶ **Theorem 4.30.** *Let* $\Delta_{\min} > 0$ *be the minimum value that satisfies*

$$\Delta_{\min} = (\max_{\tau_i \in lp(\tau_k)} C_i) + C_k + \sum_{\tau_i \in hp(\tau_k)} demand_i(\Delta_{\min}) \tag{4.27}$$

The WCRT R_k *of task* τ_k *in the problem* $1|spor, cons, fp|RT$ *is*

- $R_k \leq \Delta_{\min}$*, if* $\Delta_{\min} \leq T_k$.

Note that when $\Delta_{\min} > T_k$*, the WCRT of task* τ_k *can not be determined by this method.*

Proof. The proof is left as an exercise. ◀

For FP-NP scheduling algorithms, we need another notation to achieve tighter analysis:

$$\mathbf{FNJ}_{i,[r,r+\Delta]} = \{J_{i,j} \mid J_{i,j} \in \mathbf{FJ}_i, r_{i,j} \geq r, r_{i,j} \leq r + \Delta\} \tag{4.28}$$

That is, for a given feasible set $\mathbf{FJ}_i$ of jobs generated by a sporadic/periodic real-time task τ_i, let $\mathbf{FNJ}_{i,[r,r+\Delta)}$ be the subset of the jobs in $\mathbf{FJ}_i$ arriving in time interval $[r, r + \Delta]$. The difference between $\mathbf{FNJ}_{i,[r,r+\Delta]}$ here and $\mathbf{FRP}_{i,[r,r+\Delta)}$ in Eq. (4.13) is that the interval $[r, r + \Delta]$ is closed.

▶ **Lemma 4.31.** *The total amount of execution time of the jobs of* τ_i *that are* ***released*** *in a time interval* $[r, r + \Delta]$ *for any* $\Delta \geq 0$ *is*

$$\sum_{J_{i,j} \in \boldsymbol{FNJ}_{i,[r,r+\Delta]}} C_{i,j} \leq \left(\left\lfloor \frac{\Delta}{T_i} \right\rfloor + 1 \right) C_i \tag{4.29}$$

Proof. This comes from the definition of $\mathbf{FNJ}_{i,[r,r+\Delta]}$. Since $\mathbf{FNJ}_{i,[r,r+\Delta]}$ is also a feasible set of jobs generated by task τ_i, there are at most $\left\lfloor \frac{\Delta}{T_i} \right\rfloor + 1$ jobs in $\mathbf{FNJ}_{i,[r,r+\Delta]}$, each with execution time no more than C_i. ◀

The reason why we define the above $\mathbf{FNJ}_{i,[r,r+\Delta]}$ can be motivated by the following example. Consider three sporadic tasks $\tau_1 = \{C_1 = 2, D_1 = T_1 = 5\}$, $\tau_2 = \{C_2 = 3, D_2 = T_2 = 9\}$, and $\tau_3 = \{C_3 = 2, D_3 = T_3 = 12\}$. Suppose that these three tasks release jobs periodically from time 0. For an RM-NP schedule, at time 5, since the second job of task τ_1 is released again, the schedule should execute task τ_1 from time 5 to 7. If we analyze the demand based on $\sum_{\tau_i \in hp(\tau_3)} \left\lceil \frac{5}{T_i} \right\rceil = 5$, we may reach a wrong conclusion that task τ_3 can be executed at time 5.

Since the scheduler is non-preemptive, as long as we can safely calculate the latest starting time after a job of task τ_k arrives, this job will be finished in at most C_k time units after it starts. Since the scheduler is non-preemptive, a lower-priority job may block a higher-priority job. But, as long as a higher-priority job is already released, it is not going to be blocked by any further lower-priority job except the one that is executed when the higher-priority job arrives.

Based on the above observation, it may be possible to simply extend the critical instant theorem for preemptive fixed-priority scheduling. To know the worst-case response time of task τ_k under non-preemptive fixed-priority scheduling, we release a lower-priority job in $lp(\tau_k)$ with the longest execution time at time $-\epsilon$ and release the jobs of $hp(\tau_k)$ and τ_k periodically, starting from time 0.

We will prove that this observation is in general *correct*, but the first job of task τ_k is potentially not the job with the worst-case response time of task τ_k due to self-pushing [20], demonstrated by the following example with 4 sporadic tasks:

- $\tau_1 = \{C_1 = 3, T_1 = D_1 = 8\}$,
- $\tau_2 = \{C_2 = 3, T_2 = D_2 = 9\}$,
- $\tau_3 = \{C_3 = 3, T_3 = D_3 = 12\}$, and
- $\tau_4 = \{C_3 = 2, T_3 = D_3 = 99\}$.

Figure 4.6 demonstrates a release pattern in which a job of task τ_4 is released at time 0 and τ_1, τ_2, τ_3 release their jobs periodically, starting from time 1. According to the above schedule, the first job of τ_3 in this release pattern has better response time than the second job of τ_3. In this concrete schedule, the non-preemptive execution of the first job of τ_3 *pushes* the second jobs of τ_1 and τ_2 and *creates* more interference on the second job of τ_3.

For the rest of this section, let B_k be $\max_{\tau_i \in lp(\tau_k)} \{C_i - \epsilon\}$ for an infinitesimal ϵ.[2]

▶ **Theorem 4.32.** *Let $\Delta_{\min,np} > 0$ be the minimum value that satisfies*

$$\Delta_{\min,np} = B_k + \sum_{\tau_i \in hp(\tau_k)} \left(\left\lfloor \frac{\Delta_{\min,np}}{T_i} \right\rfloor + 1 \right) C_i \tag{4.30}$$

The WCRT R_k of task τ_k in the problem $1|spor, fp|RT$ is not upper bounded by $\Delta_{\min,np} + C_k$ even if $\Delta_{\min,np} + C_k \leq T_k$.

Proof. The above example in Figure 4.6 provides a concrete counterexample (in the discrete time domain). ◀

The consideration of the *level-k busy interval* is again needed here.

▶ **Theorem 4.33.** *For any positive integer h, let $\Delta_{\min,block,h} > 0$ be the minimum value that satisfies*

$$\Delta_{\min,block,h} = B_k + hC_k + \sum_{\tau_i \in hp(\tau_k)} \left\lceil \frac{\Delta_{\min,block,h}}{T_i} \right\rceil C_i \tag{4.31}$$

Let h^* be the minimum (positive) integer h such that $\Delta_{\min,block,h} \leq hT_k$. For any positive integer $h \in [h^*]$, let $\Delta_{\min,start,h} > 0$ be the minimum value that satisfies

$$\Delta_{\min,start,h} = B_k + (h-1)C_k + \sum_{\tau_i \in hp(\tau_k)} \left(\left\lfloor \frac{\Delta_{\min,np}}{T_i} \right\rfloor + 1 \right) C_i \tag{4.32}$$

The WCRT R_k of task τ_k in the problem $1|spor, fp|RT$ is

$$R_k = \max_{h \in [h^*]} \{\Delta_{\min,start,h} + C_k - (h-1)T_k\} \tag{4.33}$$

*Note that the evaluated worst-case response time R_k is **exact**.*

Proof. The proof is left as an exercise by extending the proof of Theorem 4.30 with the consideration of non-preemptiveness. ◀

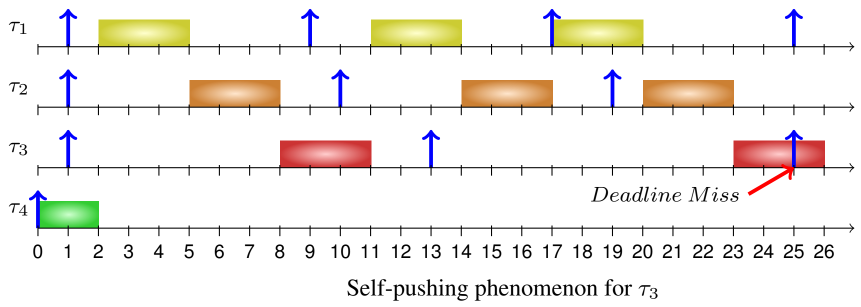


Figure 4.6: Self-pushing phenomena in non-preemptive fixed-priority scheduling

We can restrict to only looking at the first job under some (not to restrictive) conditions in the following theorem.

▶ **Theorem 4.34.** *[Yao, Buttazzo, and Bertogna [51]] The WCRT R_k of task τ_k in the problem $1|spor, fp|RT$ occurs in the first job if the task is activated at its critical time zone and the following two conditions are satisfied:*

1. *the task set is feasible under preemptive scheduling;*
2. *the relative deadlines are less than or equal to periods.*

Proof. Let $\Delta_{\min,p} > 0$ be the minimum value that satisfies

$$\Delta_{\min,p} = C_k + \sum_{\tau_i \in hp(\tau_k)} demand_i(\Delta_{\min,p}) \tag{4.34}$$

Let $\Delta_{\min,np} > 0$ be the minimum value that satisfies

$$\Delta_{\min,np} = \max_{\tau_i \in lp(\tau_k)} \{C_i - \epsilon\} + \sum_{\tau_i \in hp(\tau_k)} \left(\left\lfloor \frac{\Delta_{\min,np}}{T_i} \right\rfloor + 1 \right) C_i \tag{4.35}$$

for an infinitesimal ϵ. The WCRT R_k of τ_k in the problem $1|spor, fp|RT$ is

- $R_k \leq \Delta_{\min,np} + C_k$, if $\Delta_{\min,p} \leq T_k$ and $\Delta_{\min,np} + C_k \leq T_k$.

The proof is because the observations in the critical time zone of τ_k:

- The response time of the first job of τ_k is at most $\Delta_{\min,np} + C_k$.
- The starting time of the second job of τ_k is at most $\Delta_{\min,np} + \Delta_{\min,p} \leq \Delta_{\min,np} + T_k$. Its response time is at most $\Delta_{\min,np} + C_k$.
- etc.

◀

[2] The first term would become $\max_{\tau_i \in lp(\tau_k)}\{C_i - 1\}$ for a discrete time domain.

4.4.2 Optimality of RM-NP and DM-NP

4.5 Computational Complexity of Schedulability Tests

4.6 Remarks

4.7 Exercises

4.7.1 Incomplete Proofs in Chapter 4

▶ Exercise 4.1. Complete the proof of Theorem 4.8 by showing that $\Delta < \left(\sum_{\tau_i \in \mathbf{T}} \text{DBF}_i(\Delta)\right) + \max_{\tau_q : D_q > \Delta}\{C_q - \epsilon\}$.

▶ Exercise 4.2. Complete the proof of Theorem 4.11.
Hint: $\text{DBF}_i(t) \leq U_i t$ when $t \geq T_i$

▶ Exercise 4.3. Complete the proof of Theorem 4.14.

▶ Exercise 4.4. Prove Theorem 4.30. **Hint:** Extend the proof of Theorem 4.25. Immediately prior to time t_0', i.e., $t = t_0' - \epsilon$ for an infinitesimal ϵ, $\sigma(t)$ is either $\perp$ or a job of $lp(\tau_k)$. If it is the latter, let t_0^* be the starting time of the job executed in $\sigma(t_0' - \epsilon)$. All the jobs executed between time t_0' and $d_{k,\ell}$ are released later than t_0^*.

▶ Exercise 4.5. **(Optimality of RM)** Explain how to use Theorem 4.25 to prove that rate-monotonic scheduling is an optimal fixed-priority scheduling algorithm for the problem $1|spor, impl, prmp, fp|_{\leq}D$.

Hint: You can prove by swapping two adjacent tasks (in the priority order) if they do not follow the RM scheduling policy. As long as the schedule after swapping the priority levels of these two tasks remains feasible, we can keep swapping the tasks to convert the order into RM such that the RM schedule remains feasible.

4.7.2 Basic Exercises

▶ Exercise 4.6. Suppose that we are given the following 3 sporadic real-time tasks with implicit deadlines.

	τ_1	τ_2	τ_3
C_i	1	2	3
T_i	4	6	10

1. What are their priority levels? Is the rate-monotonic (RM) schedule feasible?

2. What happens if we change the minimum inter-arrival time of task τ_3 from 10 to 8?

▶ Exercise 4.7. Suppose that we are given the following 3 periodic real-time tasks with implicit deadlines and offsets. There are two rate-monotonic (RM) priority assignments. Prove that the task set is schedulable by one of them and not schedulable by another. (hint: a proof left to the reader in Section 4.3.3.)

	τ_1	τ_2	τ_3
C_i	3	1	6
$D_i = T_i$	8	12	12
O_i	0	10	0

▶ Exercise 4.8. **(Optimality of RM)** RM is not an optimal fixed-priority preemptive scheduling algorithm when the periodic tasks have different offsets, i.e., $1|period, impl, fp, prmp|_{\leq}D$, as presented in Section 4.3.3. Prove or disprove the optimality of RM for the problem $1|period, impl, fp, prmp|_{\leq}D$ when there are **at most two** periodic tasks.

▶ Exercise 4.9. **(Critical Instant Theorem)** Explain the critical instant theorem for uniprocessor fixed-priority scheduling in your words. As mentioned in the lecture, the critical instant theorem for uniprocessor fixed-priority scheduling is very fragile if the assumptions are not met. To apply the critical instant theorem, quite a few conditions have to be satisfied. Please indicate which of the following conditions are correct and which of them are incorrect. If a condition is incorrect, please correct it.

- The task set consists of only independent tasks.
- The task set must be *strictly* periodic.
- The scheduling algorithm is fixed-priority preemptive scheduling.
- Early completion of jobs is not possible. A job has to spin till its worst-case execution time if it finishes earlier.
- No task voluntarily suspends itself. That is, a job cannot suspend itself during its execution.
- The relative deadline of a task can be greater than its period.
- Scheduling overheads (context switch overheads) are zero.
- All periodic/sporadic tasks have zero release jitter (the time from the task arriving to it becoming ready to execute).

▶ Exercise 4.10. **(Automotive Applications)** In automotive applications periodic tasks normally have only a few possible periods. These are, for example, $\{1, 2, 5, 10, 20, 50, 100, 200, 1000\}$ milliseconds. We only look at a special subset here where the possible task periods are either 1, 2, or 5 ms.

Show that for implicit deadline task sets the utilization bound under Rate Monotonic Scheduling for such a task set (i.e., $1|spor, impl, prmp|_{\leq}D$) is 90%.

Hint: When considering the schedulability of tasks with period 5 it is sufficient to check if such a task can be scheduled at $t = 4$ or at $t = 5$.

4.7.3 Advanced Exercises

► Exercise 4.11* **(Schedulability Test under Mixed Integer Linear Programming)** Mr. Smart suggests the following schedulability test of fixed-priority scheduling for the problem $1|spor, cons, prmp, fp|_{\leq} D$. He claims that task τ_i can meet its its relative deadline under the fixed-priority scheduling if and only if the following mixed-integer linear programming has a solution.

$$C_i + \sum_{\tau_j \in hp(\tau_i)} n_j \cdot C_j \leq t \tag{4.36}$$

$$n_j \cdot T_j \geq t \qquad \forall \tau_j \in hp(\tau_i) \tag{4.37}$$

$$n_j \in \mathbb{N} \qquad \forall \tau_j \in hp(\tau_i) \tag{4.38}$$

$$0 < t \leq D_i, \tag{4.39}$$

where t is a positive variable, described in (4.35), and n_j is a positive integer number, described in (4.34). Please either explain/prove or disprove his argument.

5

Utilization Bounds and Speedup Factors

5.1 Efficient Schedulability Test of EDF-P and EDF-NP

5.2 Efficient Schedulability Test of FP-P and FP-NP

5.3 Exercises

▶ Exercise 5.1. What are the differences between the the efficient utilization-based schedulability tests and the time-demand schedulability tests for RM? Please use one example to illustrate their differences.

▶ Exercise 5.2. Given a set $\mathbf{T}$ of n independent, preemptable, and periodic tasks with implicit deadlines, they can be partitioned into $\mathbf{T}_1, \mathbf{T}_2, \ldots, \mathbf{T}_k$ task sets, in which $\bigcup_{j=1}^{k} \mathbf{T}_j$ is $\mathcal{T}$. Moreover, for $j = 1, 2, \ldots, k$, the periods of the tasks in each task set $\mathbf{T}_j$ are *simply periodic* or *harmonic*. That is, for any $\tau_i, \tau_\ell \in \mathcal{T}_j$, $\frac{T_i}{T_\ell}$ is a positive integer if $T_i \geq T_\ell$.

Prove that the task set is schedulable under rate-monotonic scheduling if

$$\sum_{i=1}^{n} U_i \leq k(2^{\frac{1}{k}} - 1),$$

Hint: Convert these n tasks to a more difficult case with only k tasks.

6

Worst-Case Execution Time (WCET)

7

Hierarchical Scheduling and Reservation Servers

8

Resource Access Protocols

9

Other Recurrent Task Models

10

Scheduling Impact Due to Self-Suspension

11

Soft Real-Time Systems

Part II

Multiprocessor Systems

12

Traditional Scheduling Theory

13

Partitioned Multiprocessor Scheduling

14

Global Multiprocessor Scheduling

15

Precedence Constraints in Multiprocessor Scheduling

16

Multiprocessor Resource Sharing with Locking Protocols

17

Multicore Systems with Shared Physical Resources

A

Appendix

Bibliography

[1] Horn W. A. Some simple scheduling algorithms. *Naval Research Logistics Quarterly*, 21(1):177–185, 1974.

[2] Tarek F. Abdelzaher, Vivek Sharma, and Chenyang Lu. A utilization bound for aperiodic tasks and priority driven scheduling. *IEEE Trans. Computers*, 53(3):334–350, 2004.

[3] Björn Andersson, Sanjoy K. Baruah, and Jan Jonsson. Static-priority scheduling on multiprocessors. In *Real-Time Systems Symposium (RTSS)*, pages 193–202, 2001.

[4] Björn Andersson and Eduardo Tovar. The utilization bound of non-preemptive rate-monotonic scheduling in controller area networks is 25%. In *IEEE Fourth International Symposium on Industrial Embedded Systems - SIES*, pages 11–18, 2009.

[5] S. Baruah and A. Burns. Sustainable scheduling analysis. In *RTSS*, pages 159–168, December 2006.

[6] Sanjoy K. Baruah, Aloysius K. Mok, and Louis E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *proceedings Real-Time Systems Symposium (RTSS)*, pages 182–190, Dec 1990.

[7] Enrico Bini, Giorgio C Buttazzo, and Giuseppe M Buttazzo. Rate monotonic analysis: the hyperbolic bound. *Computers, IEEE Transactions on*, 52(7):933–942, 2003.

[8] Enrico Bini, Andrea Parri, and Giacomo Dossena. A quadratic-time response time upper bound with a tightness property. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 13–22, 2015.

[9] Vincenzo Bonifaci, Ho-Leung Chan, Alberto Marchetti-Spaccamela, and Nicole Megow. Algorithms and complexity for periodic real-time scheduling. In *SODA*, pages 1350–1359, 2010.

[10] Yang Cai and M. C. Kong. Nonpreemptive scheduling of periodic tasks in uni- and multiprocessor systems. *Algorithmica*, 15(6):572–599, 1996.

[11] Jacques Carlier. The one-machine sequencing problem. *European Journal of Operational Research*, 11(1):42 – 47, 1982.

[12] Jian-Jia Chen, Wen-Hung Huang, and Cong Liu. k2U: A general framework from k-point effective schedulability analysis to utilization-based tests. In *Real-Time Systems Symposium (RTSS)*, pages 107–118, 2015.

[13] Jian-Jia Chen, Wen-Hung Huang, and Cong Liu. k2Q: A quadratic-form response time and schedulability analysis framework for utilization-based analysis. In *2016 IEEE Real-Time Systems Symposium, RTSS*, pages 351–362, 2016.

[14] Jian-Jia Chen, Geoffrey Nelissen, and Wen-Hung Huang. A unifying response time analysis framework for dynamic self-suspending tasks. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2016.

[15] Jian-Jia Chen, Georg von der Brüggen, Wen-Hung Huang, and Robert I Davis. On the pitfalls of resource augmentation factors and utilization bounds in real-time scheduling. In *Euromicro Conference on Real-Time Systems, ECRTS*, pages 9:1–9:25, 2017.

[16] Kuan-Hsun Chen, Georg von der Brüggen, and Jian-Jia Chen. Overrun handling for mixed-criticality support in rtems. In *Workshop on Mixed-Criticality Systems*, 2016.

[17] Houssine Chetto and Maryline Silly-Chetto. Scheduling periodic and sporadic tasks in a real-time system. *Inf. Process. Lett.*, 30(4):177–184, 1989.

[18] R. I. Davis, A. Thekkilakattil, O. Gettings, R. Dobrin, and S. Punnekkat. Quantifying the exact sub-optimality of non-preemptive scheduling. In *Real-Time Systems Symposium, 2015 IEEE*, pages 96–106, Dec 2015.

[19] R.I. Davis. On the evaluation of schedulability tests for real-time scheduling algorithms. In *WATERS*, July 2016.

[20] Robert I. Davis, Alan Burns, Reinder J. Bril, and Johan J. Lukkien. Controller area network (CAN) schedulability analysis: Refuted, revisited and revised. *Real-Time Systems*, 35(3):239–272, 2007.

[21] J. S. Deogun and V. V. Raghavan. User-oriented document clustering: A framework for learning in information retrieval. In *Proceedings of the 9th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '86, pages 157–163. ACM, 1986.

[22] M. R. Garey and D. S. Johnson. *Computers and intractability: A guide to the theory of NP-completeness*. W. H. Freeman and Co., 1979.

[23] Laurent George, Nicolas Rivierre, and Marco Spuri. Preemptive and Non-Preemptive Real-Time UniProcessor Scheduling. Research Report RR-2966, Projet REFLECS, 1996.

[24] Joel Goossens. *Scheduling of Hard Real-Time Periodic Systems with Various Kinds of Deadline and Offset Constraints*. PhD thesis, Universit Libre de Bruxelles, 1999.

[25] Ronald L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal of Applied Mathematics*, 17(2):416–429, 1969.

[26] Nan Guan, Martin Stigge, Wang Yi, and Ge Yu. Fixed-priority multiprocessor scheduling with Liu and Layland's utilization bound. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 165–174, 2010.

[27] Leslie A. Hall and David B. Shmoys. Jackson's rule for single-machine scheduling: Making a good heuristic better. *Math. Oper. Res.*, 17(1):22–35, 1992.

[28] Wen-Hung Huang and Jian-Jia Chen. Techniques for schedulability analysis in mode change systems under fixed-priority scheduling. In *Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 176–186, 2015.

[29] J. R. Jackson. Scheduling a production line to minimize maximum tardiness. Technical report, University of California, Los Angeles, 1955.

[30] Kevin Jeffay, Donald F. Stanat, and Charles U. Martel. On non-preemptive scheduling of period and sporadic tasks. In *Proceedings of the Real-Time Systems Symposium*, pages 129–139, 1991.

[31] Bala Kalyanasundaram and Kirk Pruhs. Speed is as powerful as clairvoyance. *Journal of ACM*, 47(4):617–643, July 2000.

[32] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Springer US, 1st edition, 1997.

[33] Simon Kramer, Dirk Ziegenbein, and Arne Hamann. Real world automotive benchmark for free. In *6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2015.

[34] Tei-Wei Kuo and Aloysius K. Mok. Load adjustment in adaptive real-time systems. In *IEEE Real-Time Systems Symposium*, pages 160–171, 1991.

[35] John P. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *RTSS*, pages 201–209, 1990.

[36] Joseph Y.-T. Leung and M.L. Merrill. A note on preemptive scheduling of periodic, real-time tasks. *Information Processing Letters*, 11(3):115 – 118, 1980.

[37] Joseph Y.-T. Leung and Jennifer Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Perform. Eval.*, 2(4):237–250, 1982.

[38] Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher D. Gill. Analysis of global EDF for parallel tasks. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 3–13, 2013.

[39] Jing Li, Jian-Jia Chen, Kunal Agrawal, Chenyang Lu, Christopher D. Gill, and Abusayeed Saifullah. Analysis of federated and global scheduling for parallel real-time tasks. In *26th Euromicro Conference on Real-Time Systems, ECRTS*, pages 85–96, 2014.

[40] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[41] Graham McMahon and Michael Florian. On scheduling with ready times and due dates to minimize maximum lateness. *Operations Research*, 23(3):475–482, 1975.

[42] A. K. Mok. Fundamental design problems of distributed systems for the hard-real-time environment. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1983.

[43] Mitra Nasri, Sanjoy K. Baruah, Gerhard Fohler, and Mehdi Kargahi. On the optimality of RM and EDF for non-preemptive real-time harmonic tasks. In *RTNS*, page 331, 2014.

[44] Mitra Nasri and Gerhard Fohler. Non-work-conserving non-preemptive scheduling: Motivations, challenges, and potential solutions. In *28th Euromicro Conference on Real-Time Systems, ECRTS*, pages 165–175, 2016.

[45] Eugeniusz Nowicki and Stanisław Zdrzałka. A note on minimizing maximum lateness in a one-machine sequencing problem with release dates. *European Journal of Operational Research*, 23(2):266 – 267, 1986.

[46] C. Phillips, C. Stein, E. Torng, and J. Wein. Optimal time-critical scheduling via resource augmentation. In *ACM Symposium on Theory of Computing*, pages 140–149, 1997.

[47] Michael L. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Springer Publishing Company, Incorporated, 5th edition, 2016.

[48] C. N. Potts. Technical note—analysis of a heuristic for one machine sequencing with release dates and delivery times. *Operations Research*, 28(6):1436–1441, 1980.

[49] Vijay V. Vazirani. *Approximation Algorithms*. Springer, 2001.

[50] Georg von der Bruggen, Jian-Jia Chen, and Wen-Hung Huang. Schedulability and optimization analysis for non-preemptive static priority scheduling based on task utilization and blocking factors. In *Euromicro Conference on Real-Time Systems, ECRTS*, pages 90–101, 2015.

[51] Gang Yao, Giorgio Buttazzo, and Marko Bertogna. Feasibility analysis under fixed priority scheduling with fixed preemption points. In *Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 71–80, 2010.