UNIPROCESSOR SCHEDULING ALGORITHMS. As shown in Figure 3.3, uniprocessor scheduling is part of the process of developing a multiprocessor schedule. Our ability to obtain a feasible multiprocessor schedule is therefore linked to our ability to obtain feasible uniprocessor schedules. Most of this chapter deals with this problem.

Traditional rate-monotonic (RM): The task set consists of periodic, preemptible tasks whose deadlines equal the task period. A task set of \( n \) tasks is schedulable under RM if its total processor utilization is no greater than \( n \left( 2^{1/n} - 1 \right) \).

Task priorities are static and inversely related to their periods. RM is an optimal static-priority uniprocessor scheduling algorithm and is very popular. Some results are also available for the case where a task deadline does not equal its period. See Section 3.2.1.

Rate-monotonic deferred server (DS): This is similar to the RM algorithm, except that it can handle both periodic (with deadlines equal to their periods) and aperiodic tasks. See Section 3.2.1 (in Sporadic Tasks).

Earliest deadline first (EDF): Tasks are preemptible and the task with the earliest deadline has the highest priority. EDF is an optimal uniprocessor algorithm. If a task set is not schedulable on a single processor by EDF, there is no other processor that can successfully schedule that task set. See Section 3.2.2.

Precedence and exclusion conditions: Both the RM and EDF algorithms assume that the tasks are independent and preemptible anytime. In Section 3.2.3, we present algorithms that take precedence conditions into account. Algorithms with exclusion conditions (i.e., certain tasks are not allowed to interrupt certain other tasks, irrespective of priority) are also presented.

Multiple task versions: In some cases, the system has primary and alternative versions of some tasks. These versions vary in their execution time and in the quality of output they provide. Primary versions are the full-fledged tasks, providing top-quality output. Alternative versions are bare-bones tasks, providing lower-quality (but still acceptable) output and taking much less time to execute. If the system has enough time, it will execute the primary; however, under conditions of overload, the alternative may be picked. In Section 3.2.4, an algorithm is provided to do this.

IRIS tasks: IRIS stands for increased reward with increased service. Many algorithms have the property that they can be stopped early and still provide useful output. The quality of the output is a monotonically nondecreasing function of the execution time. Iterative algorithms (e.g., algorithms that compute \( r \) or \( e \)) are one example of this. In Section 3.3, we provide algorithms suitable for scheduling such tasks.

MULTIPROCESSOR SCHEDULING. Algorithms dealing with task assignment to the processors of a multiprocessor are discussed in Section 3.4. The task assignment problem is NP-hard under any but the most simplifying assumptions. As a result, we must make do with heuristics.

Utilization balancing algorithm: This algorithm assigns tasks to processors one by one in such a way that at the end of each step, the utilizations of the various processors are as nearly balanced as possible. Tasks are assumed to be preemptible.

Next fit algorithm: The next-fit algorithm is designed to work in conjunction with the rate-monotonic uniprocessor scheduling algorithm. It divides the set of tasks into various classes. A set of processors is exclusively assigned to each task class. Tasks are assumed to be preemptible.

Bin-packing algorithm: The bin-packing algorithm assigns tasks to processors under the constraint that the total processor utilization must not exceed a given threshold. The threshold is set in such a way that the uniprocessor scheduling algorithm is able to schedule the tasks assigned to each processor. Tasks are assumed to be preemptible.

Myopic offline scheduling algorithm: This algorithm can deal with nonpreemptible tasks. It builds up the schedule using a search process.

Focused addressing and bidding algorithm: In this algorithm, tasks are assumed to arrive at the individual processors. A processor that finds itself unable to meet the deadline or other constraints of all its tasks tries to offload some of its workload onto other processors. It does so by announcing which task(s) it would like to offload and waiting for the other processors to offer to take them up.

Buddy strategy: The buddy strategy takes roughly the same approach as the focused addressing algorithm. Processors are divided into three categories: underloaded, fully loaded, and overloaded. Overloaded processors ask the underloaded processors to offer to take over some of their load.

Assignment with precedence constraints: The last task assignment algorithm takes task precedence constraints into account. It does so using a trial-and-error process that tries to assign tasks that communicate heavily with one another to the same processor so that communication costs are minimized.
**CRITICAL SECTIONS.** Certain anomalous behavior can be exhibited as a result of critical sections. In particular, a lower-priority task can make a higher-priority task wait for it to finish, even if the two are not competing for access to the same critical section. In Section 3.2.1 (in Handling Critical Sections), we present algorithms to get around this problem and to provide a finite upper bound to the period during which a lower-priority task can block a higher-priority task.

**MODE CHANGES.** Frequently, task sets change during the operation of a realtime system. We have seen in Chapter 2 that a mission can have multiple phases, each phase characterized by a different set of tasks, or the same task set but with different priorities or arrival rates. In Section 3.5, we discuss the scheduling issues that arise when a mission phase changes. We look at how to delete or add tasks to the task list.

**FAULT-TOLERANT SCHEDULING.** The final part of this chapter deals with the important problem of ensuring that deadlines will continue to be met despite the occurrence of faults. In Section 3.6, we describe an algorithm that schedules backups that are activated in the event of failure.

### 3.1.2 Notation

The notation used in this chapter will be as follows.

- \( n \) Number of tasks in the task set.
- \( e_i \) Execution time of task \( T_i \).
- \( P_i \) Period of task \( T_i \), if it is periodic.
- \( I_i \) \( k \)th period of (periodic) task \( T_i \) begins at time \( I_i + (k - 1) P_i \), where \( I_i \) is called the phasing of task \( T_i \).
- \( d_i \) Relative deadline of task \( T_i \) (relative to release time).
- \( D_i \) Absolute deadline of task \( T_i \).
- \( r_i \) Release time of task \( T_i \).
- \( h_{T(t)} \) Sum of the execution times of task iterations in task set \( T \) that have their absolute deadlines no later than \( t \).

Additional notation will be introduced as appropriate.

### 3.2 CLASSICAL UNIPROCESSOR SCHEDULING ALGORITHMS

In this section, we will consider two venerable algorithms used for scheduling independent tasks on a single processor, rate-monotonic (RM) and earliest deadline first (EDF). The goal of these algorithms is to meet all task deadlines. Following that, we will deal with precedence and exclusion constraints, and consider situations where multiple versions of software are available for the same task.

The following assumptions are made for both the RM and EDF algorithms.

- **A1.** No task has any nonpreemptable section and the cost of preemption is negligible.
- **A2.** Only processing requirements are significant; memory, I/O, and other resource requirements are negligible.
- **A3.** All tasks are independent; there are no precedence constraints.

These assumptions greatly simplify the analyses of RM and EDF. Assumption A1 indicates that we can preempt any task at any time and resume it later without penalty. As a result, the number of times that a task is preempted does not change the total workload of the processor. From A2, to check for feasibility we only have to ensure that enough processing capacity exists to execute the tasks by their deadlines; there are no memory or other constraints to complicate matters. The absence of precedence constraints, A3, means that task release times do not depend on the finishing times of other tasks.

Of course, there are also many systems for which assumptions A1 to A3 are not good approximations. Later in this chapter, we will see how to deal with some of these.

### 3.2.1 Rate-Monotonic Scheduling Algorithm

The rate-monotonic (RM) scheduling algorithm is one of the most widely studied and used in practice. It is a uniprocessor static-priority preemptive scheme. Except where it is otherwise stated, the following assumptions are made in addition to assumptions A1 to A3.
A4. All tasks in the task set are periodic.

A5. The relative deadline of a task is equal to its period.

Assumption A5 simplifies our analysis of RM greatly, since it ensures that there can be at most one iteration of any task alive at any time.

The priority of a task is inversely related to its period; if task \( T \) has a smaller period than task \( T_j \), \( T \) will have higher priority than \( T_j \). Higher-priority tasks can preempt lower-priority tasks.

**Example 3.5.** Figure 3.4 contains an example of this algorithm. There are three tasks, with \( P_1 = 2 \), \( P_2 = 6 \), \( P_3 = 10 \). The execution times are \( e_1 = 0.5 \), \( e_2 = 2.0 \), \( e_3 = 1.75 \), and \( I_1 = 0 \), \( I_2 = 1 \), \( I_3 = 3 \). Since \( P_1 < P_2 < P_3 \), task \( T_1 \) has highest priority. Every time it is released, it preempts whatever is running on the processor. Similarly, task \( T_3 \) cannot execute when either task \( T_1 \) or \( T_2 \) is unfinished.

There is an easy schedulability test for this algorithm, as follows:

If the total utilization of the tasks is no greater than \( \frac{n(2^{1/n} - 1)}{n} \), where \( n \) is the number of tasks to be scheduled, then the RM algorithm will schedule all the tasks to meet their respective deadlines. Note that this is a sufficient, but not necessary, condition. That is, there may be task sets with a utilization greater than \( n(2^{1/n} - 1) \) that are schedulable by the RM algorithm.

The \( \frac{n(2^{1/n} - 1)}{n} \) bound is plotted in Figure 3.5.

**FIGURE 3.4** Example of the RM algorithm; \( K_j \) denotes the jth release (or iteration) of Task \( T_K \).

Let us now turn to determining the necessary and sufficient conditions for RM-schedulability. To gain some intuition into what these conditions are, let us determine them from first principles for the three-task example in Example 3.5.

Assume that the task phasings are all zero (i.e., the first iteration of each task is released at time zero). Observe the first iteration of each task. Let us start with task \( T_1 \). This is the highest-priority task, so it will never be delayed by any other task in the system. The moment \( T_1 \) is released, the processor will interrupt anything else it is doing and start processing it. As a result, the only condition that must be satisfied to ensure that \( T_1 \) can be feasibly scheduled is that \( e_1 \leq P_1 \). This is clearly a necessary, as well as a sufficient, condition.

Now, turn to task \( T_2 \). It will be executed successfully if its first iteration can find enough time over \([0, P_2]\). Suppose \( T_2 \) finishes at time \( t \). The total number of iterations of task \( T_1 \) that have been released over \([0, t]\) is \( \frac{t}{P_1} \). In order for \( T_2 \) to finish at \( t \), every one of the iterations of task \( T_1 \) released in \([0, t]\) must be completed, and in addition there must be \( e_2 \) time available to execute \( T_2 \). That is, we must satisfy the condition:

\[
\text{Number of Tasks (n)}
\]

![Figure 3.5](image)
\[ t = \frac{t}{P_1} e_1 + e_2 \]

If we can find some \( t \in [0, P_2] \) satisfying this condition, we are done.

Now comes the practical question of how we check that such a \( t \) exists. After all, every interval has an infinite number of points in it, so we can't very well check exhaustively for every possible \( t \). The solution lies in the fact that 
\[ \frac{t}{P_1} \]
only changes at multiples of \( P_1 \), with jumps of \( e \). So, if we show that there exists some integer \( k \) such that \( kP_1 \geq ke_1 + e2 \) and \( kP_1 \leq P_2 \), we have met the necessary and sufficient conditions for \( T_2 \) to be schedulable under the RM algorithm. That is, we only need to check if 
\[ t = \frac{t}{P_1} e_1 + e_2 \]
for some value of \( t \) that is a multiple of \( P_1 \), such that \( t \leq P_2 \). Since there is a finite number of multiples of \( P_1 \) that are less than or equal to \( P_2 \), we have a finite check.

Finally, consider task \( T_3 \). Once again, it is sufficient to show that its first iteration completes before \( P_3 \). If \( T_3 \) completes executing at \( t \), then by an argument identical to that for \( T_2 \), we must have:
\[ t = \frac{t}{P_1} e_1 + \frac{t}{P_2} e_2 + e_3 \]
\( T_3 \) is schedulable iff there is some \( t \in [0, P_3] \) such that the above condition is satisfied. But, the right-hand side (RHS) of the above equation has jumps only at multiples of \( P_1 \) and \( P_2 \). It is therefore sufficient to check if the inequality
\[ t \geq \frac{t}{P_1} e_1 + \frac{t}{P_2} e_2 + e_3 \]
\( P_2 \) is satisfied for some \( t \) that is a multiple of \( P_1 \) and/or \( P_2 \), such that \( t \leq P_3 \).

We are now ready to present the necessary and sufficient condition in general. We will need the following additional notation:
\[ W_i(t) = \sum_{j=1}^{i} e_j \frac{t}{P_j} \]
\[ L_i(t) = \frac{W_i(t)}{t} \]
\[ L_i = \min_{0 \leq t \leq P_i} L_i(t) \]
\[ L = \max \{ L_i \} \]
\( W_i(t) \) is the total amount of work carried by tasks \( T_1, T_2, \ldots, T_i \) initiated in the interval \([0, t]\). If all tasks are released at time 0, then task \( T_i \) will complete under the RM algorithm at time \( t' \), such that \( W_i(t') = t' \) (if such a \( t' \) exists).

The necessary and sufficient condition for schedulability is as follows.

Given a set of \( n \) periodic tasks (with \( P_1 \leq P_2 \leq \ldots \leq P_n \)). Task \( T_i \) can be feasibly scheduled using RM iff \( L_i \leq 1 \).
HANDLING CRITICAL SECTIONS. In our discussions so far, we have assumed that all tasks can be preempted at any point of their execution. However, sometimes tasks may need to access resources that cannot be shared. For example, a task may be writing to a block in memory. Until this is completed, no other task can access that block, either for reading or for writing. A task that is currently holding the unsharable resource is said to be in the critical section associated with the resource.

One way of ensuring exclusive access is to guard the critical sections with binary semaphores. These are like locks. When the semaphore is locked (e.g., by setting it to 1), it indicates that there is a task currently in the critical section. When a task seeks to enter a critical section, it checks if the corresponding semaphore is locked. If it is, the task is stopped and cannot proceed further until that semaphore is unlocked. If it is not, the task locks the semaphore and enters the critical section. When a task exits the critical section, it unlocks the corresponding semaphore. For convenience, we shall say that a critical section $S$ is locked (unlocked) when we mean that the semaphore associated with $S$ is locked (unlocked).

We will assume that critical sections are properly nested. That is, if we have sections $S_1, S_2$ on a single processor, the following sequence is allowed: Lock $S_1$. Lock $S_2$. Unlock $S_2$. Unlock $S_1$, while the following is not: Lock $S_1$. Unlock $S_1$. Unlock $S_2$. Unlock $S_2$.

Everything in this section refers to tasks sharing a single processor. We assume that once a task starts, it continues until it (a) finishes, (b) is preempted by some higher-priority task, or (c) is blocked by some lower-priority task that holds the lock on a critical section that it needs. We do not, for example, consider a situation where a task suspends itself when executing I/O operations or when it encounters a page fault. The results of this section can easily be extended for this case, however (see Exercise 3.12).

It is possible for a lower-priority task $T_L$ to block a higher-priority task, $T_H$. This can happen when $T_H$ needs to access a critical section that is currently being accessed by $T_L$. Although $T_H$ has higher priority than $T_L$, to ensure correct functioning, $T_L$ must be allowed to complete its critical section access before $T_H$ can access it.

Such blocking of a higher-priority task by a lower-priority task can have the unpleasant side effect of priority inversion. This is illustrated in Example 3.15.

Example 3.15. Consider tasks $T_1, T_2, T_3$, listed in descending order of priority, which share a processor. There is one critical section $S$ that both $T_1$ and $T_3$ use. See Figure 3.14. $T_3$ begins execution at time $t_0$. At time $t_1$, it enters its critical section $S$. $T_1$ is released at time $t_2$ and preempts $T_3$. It runs until $t_3$, when it tries to enter the critical section $S$. However, $S$ is still locked by the suspended task $T_3$. So, $T_1$ is suspended and $T_3$ resumes execution. At time $t_6$, task $T_2$ is released. $T_2$ has higher priority than $T_3$, and so it preempts $T_3$. $T_2$ does not need $S$ and runs to completion at $t_8$. After $T_2$ completes execution at $t_8$, $T_3$ resumes and exits critical section $S$ at $t_9$. $T_1$ can now preempt $T_3$ and enter the critical section.

Notice that although $T_2$ is of lower priority than $T_1$, it was able to delay $T_1$ indirectly (by preemting $T_3$, which was blocking $T_1$). This phenomenon is known as priority inversion. Ideally, the system should have noted that $T_1$ was waiting for access, and so $T_2$ should not have been allowed to start executing at $t_6$.
The use of priority inheritance allows us to avoid the problem of priority inversion. Under this scheme, if a higher-priority task \( T_h \) is blocked by a lower-priority task \( T_l \) (because \( T_l \) is currently executing a critical section needed by \( T_h \)), the lower-priority task temporarily inherits the priority of \( T_h \). When the blocking ceases, \( T_l \) resumes its original priority. The protocol is described in Figure 3.15. Example 3.16 shows how this prevents priority inversion from happening.

**Example 3.16.** Let us return to Example 3.15 to see how priority inheritance prevents priority inversion.

At time \( t_3 \), when \( T_3 \) blocks \( T_1 \), \( T_3 \) inherits the higher priority of \( T_1 \). So, when \( T_2 \) is released at \( t_4 \), it cannot interrupt \( T_3 \). As a result, \( T_1 \) is not indirectly blocked by \( T_2 \).

1. The highest-priority task \( T \) is assigned the processor. \( T \) relinquishes the processor whenever it seeks to lock the semaphore guarding a critical section that is already locked by some other job.
2. If a task \( T_j \) is blocked by \( T_i \) (due to contention for a critical section) and \( T_j > T_i \), task \( T_j \) inherits the priority of \( T_i \) as long as it blocks \( T_i \). When \( T_i \) exits the critical section that caused the block, it reverts to the priority it had when it entered that section. The operations of priority inheritance and the resumption of previous priority are indivisible.
3. Priority inheritance is transitive. If \( T_j \) blocks \( T_i \), which blocks \( T_1 \) (with \( T_1 > T_2 > T_3 \)), then \( T_j \) inherits the priority of \( T_i \) through \( T_2 \).
4. A task \( T_j \) can preempt another task \( T_i \) if \( T_j \) is not blocked and if the current priority of \( T_j \) is greater than that of the current priority of \( T_i \).

Unfortunately, priority inheritance can lead to deadlock. This is illustrated by Example 3.17.

**Example 3.17.** Consider two tasks \( T_1 \) and \( T_2 \), which use two critical sections \( S_1 \) and \( S_2 \). These tasks require the critical sections in the following sequence:

- \( T_1: \) Lock \( S_1 \), Lock \( S_2 \), Unlock \( S_2 \), Unlock \( S_1 \).
- \( T_2: \) Lock \( S_2 \), Lock \( S_1 \), Unlock \( S_1 \), Unlock \( S_2 \).

Let \( T_1 > T_2 \), and suppose \( T_2 \) starts execution at \( t_0 \). At time \( t_0 \), \( T_2 \) locks \( S_2 \). At time \( t_1 \), \( T_1 \) is initiated and it preempts \( T_2 \) owning to its higher priority. At time \( t_2 \), \( T_1 \) locks \( S_1 \). At time \( t_3 \), \( T_1 \) attempts to lock \( S_2 \), but is blocked because \( T_2 \) has not finished with it. \( T_1 \), which now inherits the priority of \( T_2 \), starts executing. However, when at time \( t_4 \) it tries to lock \( S_1 \), it cannot do so since \( T_1 \) has a lock on it. Both \( T_1 \) and \( T_2 \) are now deadlocked.

There is another drawback of priority inheritance. It is possible for the highest-priority task to be blocked once by every other task executing on the same processor. (The reader is invited in Exercise 3.8 to construct an example of this.)

To get around both problems, we define the priority ceiling protocol. The priority ceiling of a semaphore is the highest priority of any task that may lock it. Let \( P(T) \) denote the priority of task \( T \), and \( P(S) \) the priority ceiling of the semaphore of critical section \( S \).

**Example 3.18.** Consider a three-task system \( T_1, T_2, T_3 \) with \( T_1 > T_2 > T_3 \). There are four critical sections, and the following table indicates which tasks may lock which sections, and the resultant priority ceilings.

<table>
<thead>
<tr>
<th>Critical Section</th>
<th>Accessed by</th>
<th>Priority ceiling</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S_1 )</td>
<td>( T_1, T_2 )</td>
<td>( P(T_1) )</td>
</tr>
<tr>
<td>( S_2 )</td>
<td>( T_1, T_2, T_3 )</td>
<td>( P(T_1) )</td>
</tr>
<tr>
<td>( S_3 )</td>
<td>( T_3 )</td>
<td>( P(T_3) )</td>
</tr>
<tr>
<td>( S_4 )</td>
<td>( T_2, T_3 )</td>
<td>( P(T_2) )</td>
</tr>
</tbody>
</table>

The priority ceiling protocol is the same as the priority inheritance protocol, except that a task can also be blocked from entering a critical section if there exists any semaphore currently held by some other task whose priority ceiling is greater than or equal to the priority of \( T \).

**Example 3.19.** Consider the tasks and critical sections mentioned in Example 3.18. Suppose that \( T_2 \) currently holds a lock on \( S_2 \), and task that \( T_1 \) is initiated. \( T_1 \) will be blocked from entering \( S_1 \) because its priority is not greater than the priority ceiling of \( S_1 \).
The priority ceiling protocol is specified in Figure 3.16. The key properties of the priority ceiling protocol are as follows:

P1. The priority ceiling protocol prevents deadlocks.

P2. Let \( B_i \) be the set of all critical sections that can cause the blocking of task \( T \) and \( t(x) \) be the time taken for section \( x \) to be executed. Then, \( T \) will be blocked for at most \( \max_{x \in B_i} t(x) \).

1. The highest-priority task, \( T \), is assigned the processor. \( T \) relinquishes the processor (i.e., it is blocked) whenever it seeks to lock the semaphore guarding a critical section which is already locked by some other task \( Q \) (in which case it is said to be blocked by task \( Q \)), or when there exists a semaphore \( S' \) locked by some other task, whose priority ceiling is greater than or equal to the priority of \( T \). In the latter case, let \( S^* \) be the semaphore with the highest priority among those locked by some other tasks. We say that \( T \) is blocked on \( S^* \), and by the task currently holding the lock on \( S^* \).

2. Suppose \( T \) blocks one or more tasks. Then, it inherits the priority of the highest-priority task that it is currently blocking. The operations of priority inheritance and resumption of previous priority are indivisible.

3. Priority inheritance is transitive.

4. A task \( T_1 \) can preempt another task \( T_2 \) if \( T_2 \) does not hold a critical section which \( T_1 \) currently needs, and if the current priority of \( T_1 \) is greater than that of the current priority of \( T_2 \).

**Figure 3.16** The priority ceiling protocol

### 3.2.2 Preemptive Earliest Deadline First (EDF) Algorithm

A processor following the EDF algorithm always executes the task whose absolute deadline is the earliest. EDF is a *dynamic-priority* scheduling algorithm; the task priorities are not fixed but change depending on the closeness of their absolute deadline. EDF is also called the deadline-monotonic scheduling algorithm.

**Example 3.20.** Consider the following set of (aperiodic) task arrivals to a system.

<table>
<thead>
<tr>
<th>Task</th>
<th>Arrival time</th>
<th>Execution time</th>
<th>Absolute deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>0</td>
<td>10</td>
<td>30</td>
</tr>
<tr>
<td>T2</td>
<td>4</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>T3</td>
<td>5</td>
<td>10</td>
<td>25</td>
</tr>
</tbody>
</table>

When \( T_1 \) arrives, it is the only task waiting to run, and so starts executing immediately. \( T_2 \) arrives at time 4; since \( d_2 < d_1 \), it has higher priority than \( T_1 \) and preempts it. \( T_3 \) arrives at time 5; however, since \( d_3 > d_2 \), it has lower priority than \( T_2 \) and must wait for \( T_2 \) to finish. When \( T_2 \) finishes (at time 7), \( T_3 \) starts (since it has higher priority than \( T_1 \)). \( T_3 \) runs until 17, at which point \( T_1 \) can resume and run to completion.

In our treatment of the EDF algorithm, we will make all the assumptions we made for the RM algorithm, except that the tasks do not have to be periodic.

EDF is an optimal uniprocessor scheduling algorithm. That is, if EDF cannot feasibly schedule a task set on a uniprocessor, there is no other scheduling algorithm that can.

If all the tasks are periodic and have relative deadlines equal to their periods, the test for task-set schedulability is particularly simple:

If the total utilization of the task set is no greater than 1, the task set can be feasibly scheduled on a single processor by the EDF algorithm.

There is no simple schedulability test corresponding to the case where the relative deadlines do not all equal the periods; in such a case, we actually have to develop a schedule using the EDF algorithm to see if all deadlines are met over a given interval of time. The following is a schedulability test for EDF under this case.
Define \( u = \sum_{i=1}^{n} \left( \frac{e_i}{P_i} \right) \), \( d_{\text{max}} = \max_{1 \leq i \leq n} \left\{ d_i \right\} \) and \( P = \text{lcm}(P_1, \ldots, P_n) \). (Here "lcm" stands for least common multiple.) Define \( h_T(t) \) to be the sum of the execution times of all tasks in set \( T \) whose absolute deadlines are less than \( t \). A task set of \( n \) tasks is not EDF-feasible iff

- \( u > 1 \) or
- there exists \( t < \min P + d_{\text{max}} , \frac{u}{1-u} \max_{1 \leq i \leq n} \left\{ P_i - d_i \right\} \)

such that \( h_T(t) > t \).