3 Real-Time Scheduling

Summary: *In previous chapter we presented basic services provided by a real-time operating system. We mentioned that a real-time scheduler is a central part of a RTOS since it determines in which order and with which resources assigned tasks will execute, such that they meet their real-time requirements. However, we did not talk about how the actual scheduling of tasks is performed, which is the objective of this chapter. Here we will discuss several aspects regarding the scheduling algorithms for real-time tasks execution, such as modeling of timing constraints for the tasks, different classes of scheduling algorithms, scheduling with shared resources involved, as well as different timing analysis techniques to guarantee timing behavior of a real-time system.*

3.1 Learning objectives of this chapter

After reading this chapter you should be able to

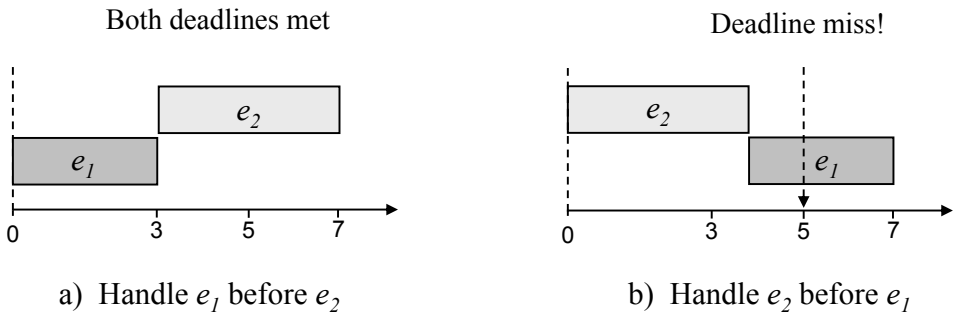
- Understand basic concepts needed for real-time scheduling, such as task timing attributes (deadlines, release times, periods, etc), schedulability, feasibility and system utilization.
- Use different scheduling algorithms to schedule tasks in both event-triggered and time-triggered systems.
- Understand the consequences of having shared resources in the systems on schedulability, and be able to use different resource allocation protocols to handle shared resources.
- Analyze, predict and guarantee timing behavior of a real-time system in offline and online scheduled systems.

3.2 Introduction to real-time scheduling

In Chapter 1, we defined a real-time system as one that reacts to external events, performs computation based on them, and delivers an answer within a specified time interval. Based on this, we can say in general that events are controlling the execution of software. Some events occur at regular time intervals, e.g., a hardware clock that generates clock interrupts each 10 milliseconds, and some events that can occur anytime, e.g., inflation of an airbag in a car upon collision. Based on this, we divided real-time systems into time-triggered and event-triggered systems, see chapters 1 and 2 for discussion on those.

The order in which the system handles events is important. Consider, for example, two events e_1 and e_2 that occur simultaneously. Assume that it takes 3 ms for the system to handle e_1 and 4 ms to handle e_2. Assume also that there is a requirement that the task that handles e_1 must finish within 5 ms, and the one that handles e_2 has a deadline of 7 ms. If the system handles event e_1 before e_2, both of them will be served on time, see Figure 1-a. However, if the system

takes care of e_2 prior to e_1, then the task that handles e_1 will miss its deadline, as depicted in Figure 1-b.



a) Handle e_1 before e_2 b) Handle e_2 before e_1

Figure 1: Order of execution is important.

As we can see from the example above, it does matter in which order the tasks are executed. The process of deciding the execution order of real-time tasks is called real-time *scheduling*. A real-time *schedule* is an assignment of tasks to the processor such that all tasks meet their timing constraints. We have mentioned scheduling several times in chapters 1 and 2, but we did not talk about any particular scheduling algorithms so far. In this chapter we will describe several scheduling algorithms, but first we will start by introducing the framework needed to explain the algorithms and the differences between them.

3.3 Task Model

A *task model* is an abstract model that can express and specify different system requirements on tasks. It should be able to express the different task types the system can handle, their timing constraints and their interaction with each other. Moreover, a task model should be able to express resource allocation needs of the tasks, the type of communication and synchronization between tasks, as well as the order of execution for the tasks. Next we describe the task parameters that we will use in our task model.

Task parameters

The task parameters of a task model can be static and dynamic, depending on whether they change during run-time of the system or not.

Static task parameters describe the properties of a task that are independent of the other tasks in the system, and they do not change during task execution. They are extracted directly for the system specification, e.g., by a careful analysis of the external process to be controlled. An example of a static task parameter is the period time of a task (already mentioned in chapters 1 and 2). Another example is the worst-case execution time of a task, which is the longest possible execution time of a task on a specific hardware platform.

Dynamic task parameters describe properties that could change during the execution of the task, i.e., they can differ between the instances of the same task (see chapter 2 for definition of task instances). They are dependent of the system scheduler and the properties of the other

tasks in the system. For example, if we use a scheduler that always prioritizes the task that has shortest time left to its deadline, then the instances of the same task will get different priorities depending on the current status of the ready queue (sometimes there will be other task instances that are more urgent to execute and hence get higher priority). Some examples of dynamic task parameters are the starting and the finishing times of instances, their actual execution time (that can vary based on the input data and possible execution paths), and the blocking time of a task, i.e., the time a task needs to wait for a resource, taken by some other task, to be free.

Note that static task parameters of a task remain unchanged for each instance of the task while the dynamic parameters can vary between different instances of the same task.

We will now give precise definition the most important task parameters, which will be used in the remainder of this book.

Definition 1: Period time

The time interval between two consecutive invocations of a periodic task.

Period time specifies how often a task wants to repeat its execution. Note "wants to repeat" instead "will repeat": a task instance that gets ready to execute will not necessarily be granted to execute immediately – there can be higher priority tasks in the ready queue. So, the period time specifies how often a task is invoked (activated), not how often it executes (see jitter definition in chapter 2).

Another term commonly used for real-time system is the deadline. In daily life, we use deadlines to specify how long time do we have to do things, i.e., catch a flight, get to the meeting, finish report, etc. Similarly, we use deadlines in real-time systems to specify an upper bound on finishing times for tasks.

Definition 2: Deadline

The latest point in time a task has to finish its computation and deliver an answer.

A deadline of a task can be specified based on a starting time of the system, called *absolute deadline*, or relatively to the activation of the task, called *relative deadline*. They express the same thing and can easily be converted to each other; if a task gets ready at time t, then *absolute deadline* = t + *relative deadline* holds. We will mostly use relative deadline in the remainder of this book. Therefore, if nothing else is specified, deadline refers to the relative deadline.

A task deadline can be hard, soft, and firm, depending on the criticality of timely result delivery. A *hard* deadline must be fulfilled; otherwise the computation delivered by the task is not useful for the system any more. Hard deadlines are usually used in safety-critical application, where the penalty of missing a deadline is very high and can possibly cause loss of human lives, e.g., an airbag that does not inflate on time. A *soft* deadline, on the other hand, can be occasionally missed without severe consequences for the systems. Missed soft deadlines will result in system performance degradation, e.g., a video frame is displayed late causing the quality of watched video to be temporarily decreased, but the system will be able to proceed with the execution after the deadline miss. A *firm* deadline is something in between soft and hard deadlines. If the firm deadline is missed, the computation performed by the task will not be useful for the system, but the system will not be put in a dangerous situation (as with missed hard deadline). Usually, there will be an acceptance test performed by the system for all tasks with firm deadlines, i.e., the system will check first if there are enough available resources for serving tasks with firm deadline before accepting them to execute.

Here are the definitions of some other tasks parameters that we will use in the book:

Definition 3: Arrival time

The time when a task instance gets activated (becomes ready to execute).

Definition 4: Offset

The time interval that specifies how long the system should delay the activation of a task (i.e., its arrival time) relatively to the start of the period.

Definition 5: Start time

The time when a task instance start to run, i.e., the task enters the executing state.

Definition 6: Finishing time

The time when a task instance has completed its execution.

Definition 7: Response time

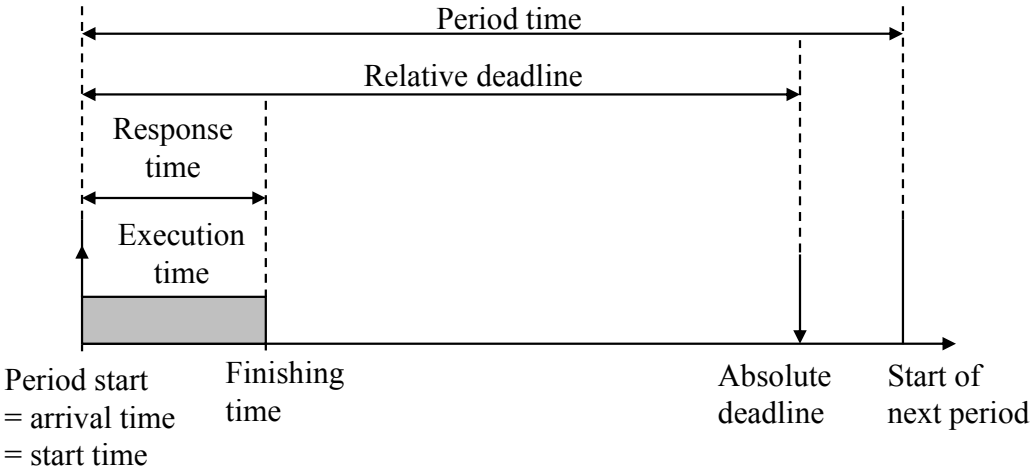
The time interval between the arrival time and the finishing time of a task instance.

Definition 8: Execution time

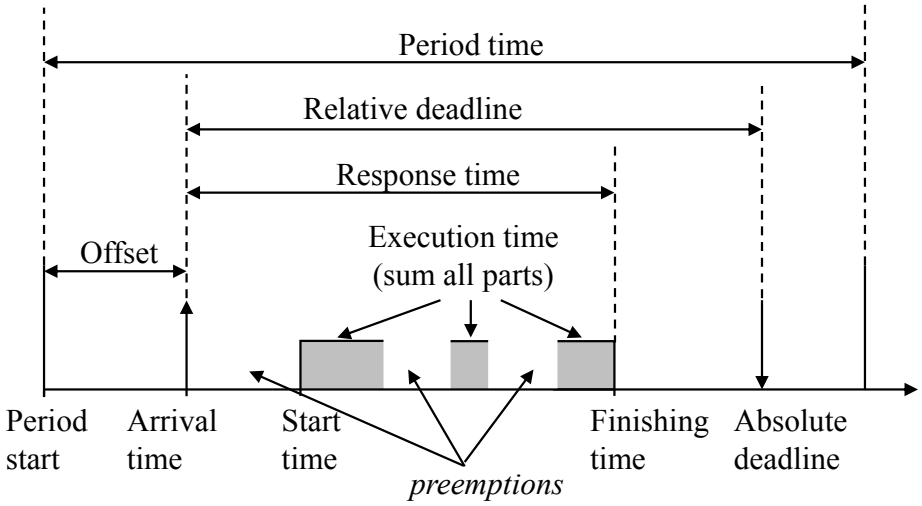
The time it takes for a task instance to finish its execution without any interruptions by other tasks. The longest execution time of all instances is called the worst-case execution time of a task.

Figure 2 gives an overview of most common task parameters for a periodic task, both with and without offsets and preemptions. For example, we can see in the Figure 1-a that the

arrival time and the start time are both equal to the start of the period; the task will become ready at the beginning of each new period and it will immediately start to execute. However, if we use offsets and allow preemptions, as illustrated in Figure 2-b, then the arrival time will be different from the period start. Similarly, since we allow preemptions, the task instance might not be able to start to execute as soon as it becomes ready, hence the start time will be later than the arrival time.



a) <u>without</u> offset and without preemptions



b) <u>with</u> offset and with preemptions

Figure 2: Task parameters for a periodic task

In the case without offsets and preemptions, the response time of a task instance will be equal to its execution time. However, if we allow preemptions, the response time will be larger than the execution time because the response time includes all preemptions that occurred during the execution of the task instance.

It is important to remember that the response time include preemptions from high priority tasks, while the execution time does not. For execution time, it does not matter if the task instance gets preempted or not, it will remain the same. For example, if it is takes 20 ms to execute a task instance on a certain processor, and if the instance gets preempted after 15 ms of execution for 3 ms, it will continue to run for additional 5 ms when it gets activated again. In total, the execution time will be 15+5=20, because the actual time that the task instance spent in the executing state is 20 ms. However, the response time will be 15+3+5=23 ms, i.e., it will include the time that the task was preempted.

We use the following notation for task parameters in our task model:

- τ_i – periodic task *i* (Greek symbol τ is pronounced as "tau")
- τ_i^j – the j^{th} invocation of τ_i

Parameters that characterize a task τ_i:

- T_i – period time of the task
- D_i – relative deadline of the task
- $WCET_i$ – worst-case execution time of the task
- O_i – offset of the task
- R_i – response time (the longest possible) of the task

Parameters that characterize a task instance τ_i^j:

- a_i^j – arrival time of the instance
- st_i^j – start time of the instance
- ft_i^j – finishing time of the instance
- c_i^j – actual execution time of the instance
- r_i^j – actual response time of the instance
- d_i^j – absolute deadline of the instance

Now it gets easier to express the dependencies between the parameters, since we can use mathematic expressions instead of words. For example, the relation between the absolute and relative deadline can be expressed as:

$$d_i^j = a_i^j + D_i$$

Same way, the actual response time of a task instance can be obtained as:

$$r_i^j = ft_i^j - a_i^j$$

The worst-case response time of a task, R_i, is the maximum of all response times of all individual task instances. We will see later in this chapter how R_i is calculated.

Task types

The most common task type in real-time systems is periodic tasks, but there are some other types as well, such as aperiodic and sporadic tasks. Here we describe them in terms of their parameters.

Periodic tasks – A periodic task τ_i can be described by four parameters (offset, execution time, deadline, and period time):

$$\tau_i = \{O_i, C_i, D_i, T_i\}$$

As mentioned before in chapter 2, a periodic task consists of an infinite sequence of identical instances or jobs that are activated within regular time periods, which are calculated as:

$$a_i^0 = O_i$$
$$\forall j > 0, a_i^j = a_i^{j-1} + T_i$$

Another way to express the arrival times of the instances is (except the first one which is equal to offset):

$$\forall j > 0, a_i^j = O_i + (j-1)T_i$$

If there is no offset specified, the task instances will be activated at times *0, T_i, $2T_i$, $3T_i$, ...*

Sporadic tasks – this type of tasks is used to handle events that arrive at the system at arbitrary points in time, but with defined maximum frequency. Just like periodic tasks, they are invoked repeatedly with a (non-zero) lower bound on the duration between consecutive invocations, but the difference is that a sporadic task may invoke its instances irregularly. Before runtime, it is known what is the minimum time between consecutive instances, called *minimum inter-arrival time* (T^{min}), but the actual time between arrivals of instances is not know until runtime, i.e., it becomes known first when the instance arrives.

A sporadic task is usually expressed with three parameters (execution time, deadline and minimum inter-arrival time):

$$\tau_i = \{C_i, D_i, T_i^{min}\}$$

The following must hold for all instances of a sporadic task:

$$\forall j > 0, a_i^j \geq a_i^{j-1} + T_i^{min}$$
$$\forall j > 0, d_i^j = a_i^j + D_i$$

Note "greater or equal" for arrival times, which means that the next instance can be invoked earliest after T^{min} time units, counted from the arrival of the current instance. The exact arrival times are not known (until when they actually occur at runtime).

Note the difference from periodic tasks, where we know before runtime that the instances will be invoked with exactly T time units in between. Periodic tasks are activated with regular

periodicity by the system clock, while sporadic tasks usually wait for some event which in general is not periodic (e.g., arrival of a data packet from a network).

Aperiodic tasks – Aperiodic tasks are tasks for which we do not know anything about the frequency of their instances. They usually arise from asynchronous events outside the system, such as operator requests, or an emergency button pressed; we cannot possible predict how often the task will get activated.

An aperiodic task is characterized by two parameters (execution time and deadline):

$$\tau_i = \{C_i, D_i\}$$

Arrival times for the instances are unknown, because an aperiodic task does not have any guaranteed minimum time between its invocations. The arrival times will be known at runtime only, when the event that triggers the task occurs.

Figure 3 illustrates all three task types. As we can see in the figure, the instances of a periodic task will be invoked regularly; the sporadic instances will be invoked with a minimum time in between, while the activation times of aperiodic instances will be random.

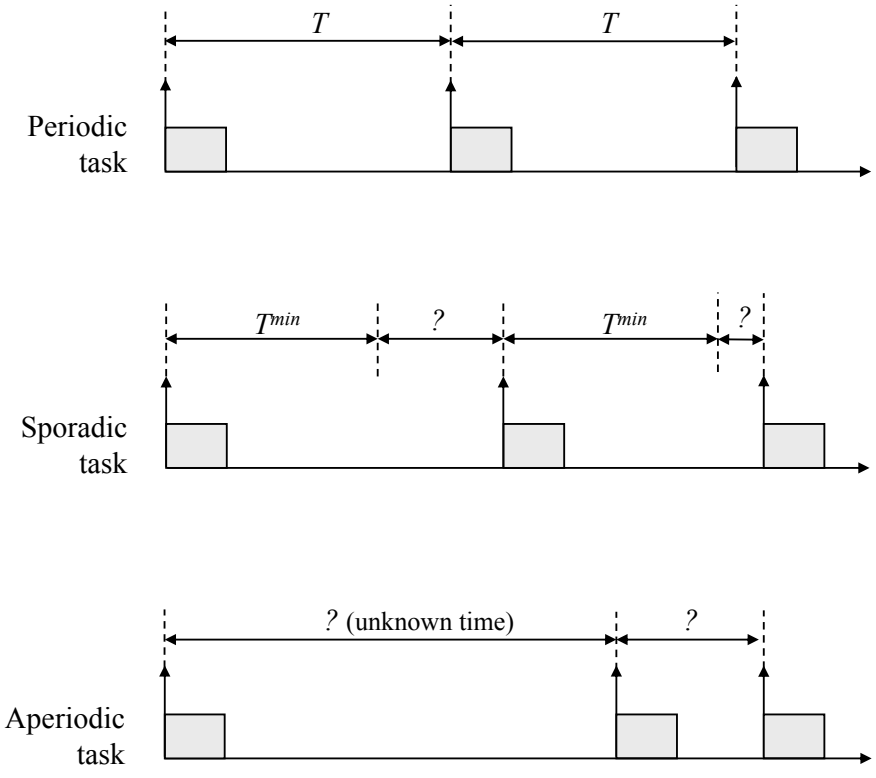


Figure 3: Task types

3.4 Schedulability and feasibility

A *feasible* schedule for a task set is schedule in which all tasks in the set are executed within their deadlines and all the other constraints, if any, are met. A task set is said to be *schedulable* if there exists a feasible schedule for the set.

A *schedulability test* is used to determine if the task set is schedulable or not with a certain scheduling algorithm. The test can be sufficient and/or necessary. A *sufficient* schedulability test that is passed will guarantee that the task set is schedulable. However, if the test is not passed, we cannot make any conclusions about the schedulability; the task set can be schedulable or not. In other words, a sufficient schedulability test can only give us a reliable answer if it is fulfilled. A *necessary* schedulability test, on the other hand, provides a reliable answer only if not passed, i.e., if the test is not passed, the task set is not schedulable.

A schedulability test that is both sufficient and necessary at the same time is called *exact*, and it can guarantee an answer whether the task set is schedulable or not. If passed, the task set is schedulable, if not passed, the set is not schedulable. The objective is to find an exact test for each schedulability algorithm.

3.5 Hyperperiod

In chapter 2, we mentioned that a real-time schedule with periodic tasks will repeat itself after a certain time, hyperperiod, which is equal to the least common multiple (LCM) of all task periods in the schedule. Since we will be using hyperperiods a lot in this book, here we give a more formal definition of it, and an example.

Let $T = \{\tau_1, \tau_2, \tau_3, \ldots, \tau_n\}$ be a set of n periodic tasks. Then, the hyperperiod H of the task set is defined as:

$$H(T) = LCM(\tau_i \epsilon\, T), 1 \leq i \leq n$$

This means that the execution pattern of a schedule that consists of tasks from T will repeat itself after H time units. Here is an example:

Example: Assume three periodic tasks τ_1, τ_2, and τ_3, with periods 3, 4, and 6 respectively. Calculate hyperperiod of the task set.

The hyperperiod is calculated as:

$$H = LCM(3,4,6) = 12$$

This means that the execution pattern of the tasks between time 0 and 12 will be the same as the execution pattern between time 12 and 24, which is the same as for time between 24 and 36, and so on. This is illustrated in Figure 4 (for the case where τ_1 has the highest priority, and τ, the lowest, but the ideas is the same for all priority assignments).

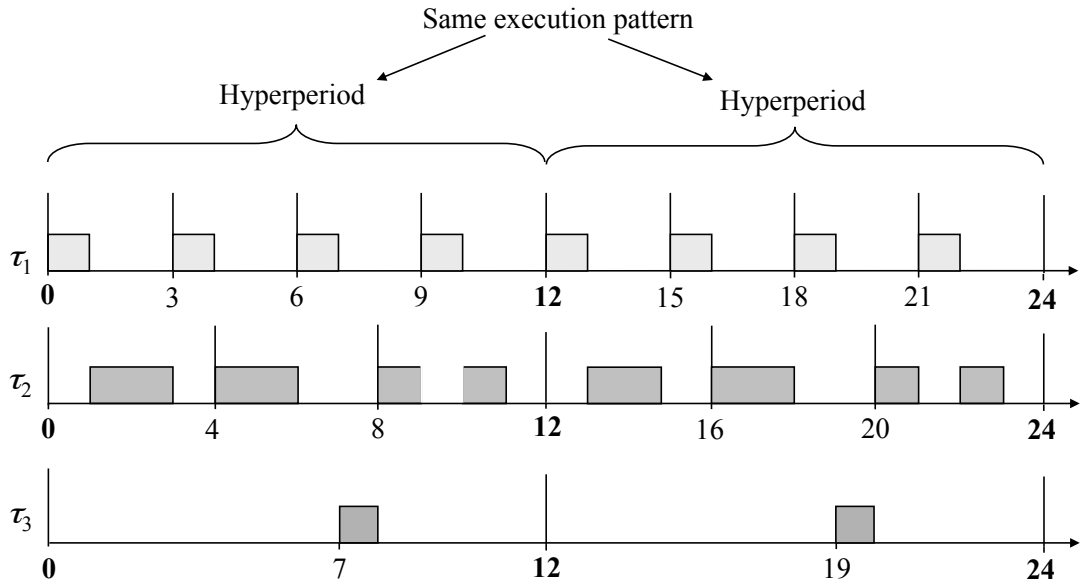


Figure 4: Example hyperperiod for three periodic task

As we can see in the figure, all three tasks are released simultaneously at times *0, H, 2H, 3H*, and so on. We also see that the schedule at, for example, time 6 is the same as the one at time 6+*H*=18.

We can conclude by saying that a scheduling algorithm will always try to find a feasible schedule within one hyperperiod, and if such schedule exists, it will also be feasible in all consecutive hyperperiod.

3.6 Processor Utilization Factor

We continue in defining parts of our system model needed for presentation and evaluation of different scheduling algorithms. One important term in this context is the *processor utilization factor*, *U*.

The processor utilization tells us how much of the processor time is needed for execution of tasks, and it is given as a number between 0 and 1 (however, it is usually expressed as percentage, 0%-100%). If the utilization is larger than 100%, then we can be sure that the task set is not schedulable, because the usage of CPU cannot be larger than 100%. We will first show how the utilization is calculated for one task, and then we will extend the equation for a whole task set.

The utilization factor for a task τ_i tells us how much of the processor time is used to execute $\tau_{i.}$, and it is calculated as:

$$U_i = \frac{C_i}{T_i}$$

Assume, for example, a periodic task τ_1 that has execution time 2 and period time of 10 clock ticks, as illustrated in Figure 5.

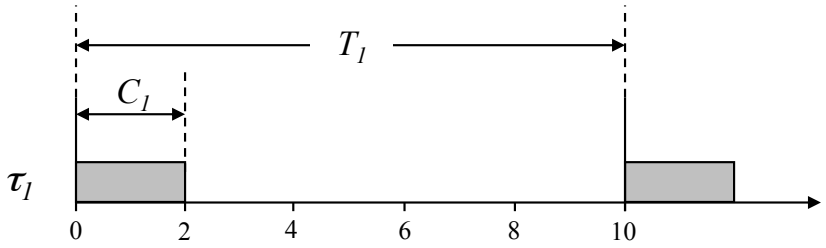


Figure 5: Example utilization factor for one task

We can see that above that each instance of τ1 uses 2 ticks of the processor time to execute, while the remaining 8 ticks are idle. Hence, the utilization for the task above is:

$$U_1 = \frac{C_1}{T_1} = \frac{2}{10} = 0.2$$

In other words, each instance of τ_1 will increase the processor load by 20%.

Given a *set* of periodic tasks, the processor utilization factor is the fraction of processor time spent in the execution of the entire task set, and it is calculated as the sum of utilizations for each task in the set:

$$U = \frac{C_1}{T_1} + \frac{C_2}{T_2} + \frac{C_3}{T_3} + \cdots \frac{C_n}{T_n} = \sum_{i=1}^{n} \frac{C_i}{T_i} \quad (1)$$

Example: Calculate utilization for the following task set:

Task	C_i	T_i
τ_1	1	10
τ_2	1	4
τ_3	1	2

Answer:

$$U = \frac{C_1}{T_1} + \frac{C_2}{T_2} + \frac{C_3}{T_3} = \frac{1}{10} + \frac{1}{4} + \frac{1}{2} = \frac{17}{20} = 0.85$$

The processor utilization when executing the three tasks will be 85%. This can be also seen by drawing the execution trace of the tasks for one hyper period. The case in which τ_3 has the highest priority, and τ_1 the lowest one is illustrated in Figure 6. We can see that there will be three idle (unused) slots in the schedule, i.e., 17 slots out of 20 possible slots are used. This means that the system utilization is U=17/20=0.85, which is the same as we obtained by using the equation (1).

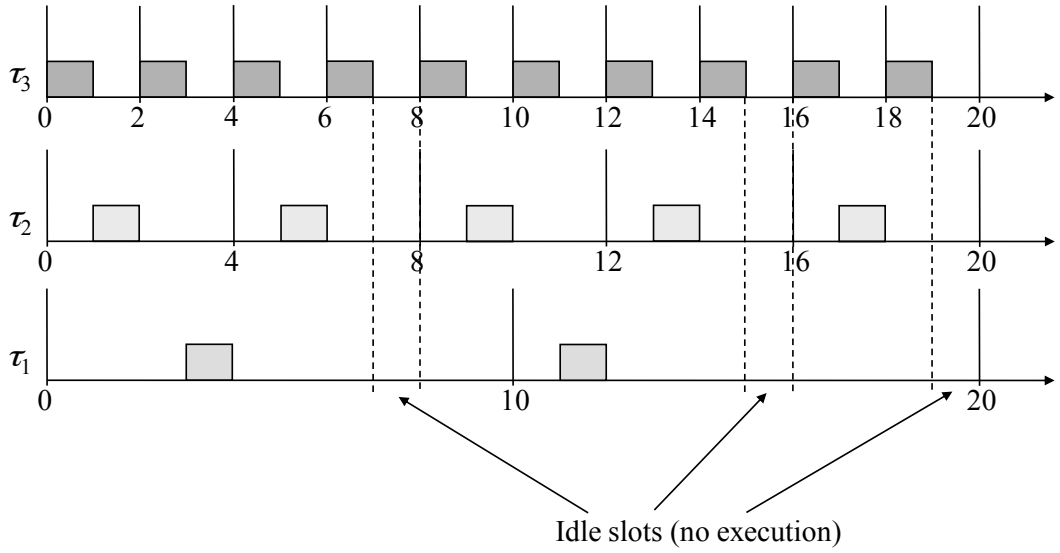


Figure 6: Example utilization of three tasks

In general, we can say that the utilization can be improved by increasing task execution times or decreasing task periods. Note that if $U>1$ for a task set, and then the set is not schedulable, i.e., the processor load will be higher than 100% and tasks will start to miss their deadlines.

3.7 Classification of scheduling algorithms

Real-time scheduling can be divided into three parts:

- *Configuration* – to decide before start of the system decide which information will be provided to the system under run-time, e.g., task priorities.
- *Run-time dispatching* – to decide how task switching will be performed at runtime, e.g., according to some specific algorithm.
- *Analysis* – to provide guarantees before runtime that all timing constraints of the tasks involved will be fulfilled. To be able to perform the analysis, we must know which configuration and which run-time dispatching algorithm are used.

The main difference between scheduling algorithms is how much of each part above is used.

On a high level, real-time scheduling can be categorized into *online* and *offline* scheduling. In online scheduling, all scheduling decisions are taken at run-time, while the system is executing. That is why this type of scheduling is also known as *run-time* scheduling. Offline scheduling, however, is performed before the system starts to operate, hence is it also called *pre-run-time* scheduling.

In online scheduling, all scheduling decisions are based on the task priorities; the task with the highest priority among all currently ready tasks is scheduled to execute first. The main difference between online scheduling algorithms is how and when the priorities are assigned to tasks. Based on this, online scheduling can be further divided into *static priority* scheduling and *dynamic priority* scheduling.

Static priority scheduling, also known as *fixed priority* scheduling, is, as the name suggests, based on fixed priorities that are assigned to tasks before run-time. The priorities are then used at run-time to make scheduling decisions. They do not change at run-time, i.e., all instances of a task will have the same priority relative to the other tasks in the systems.

In dynamic priority scheduling, task priorities may change under runtime. Different instances of the same task may have different priorities, depending on current situation of the system, and the used priority policy.

Figure 7 gives an overview of a simple classification of real-time scheduling. We will now describe each of the algorithms in the figure.

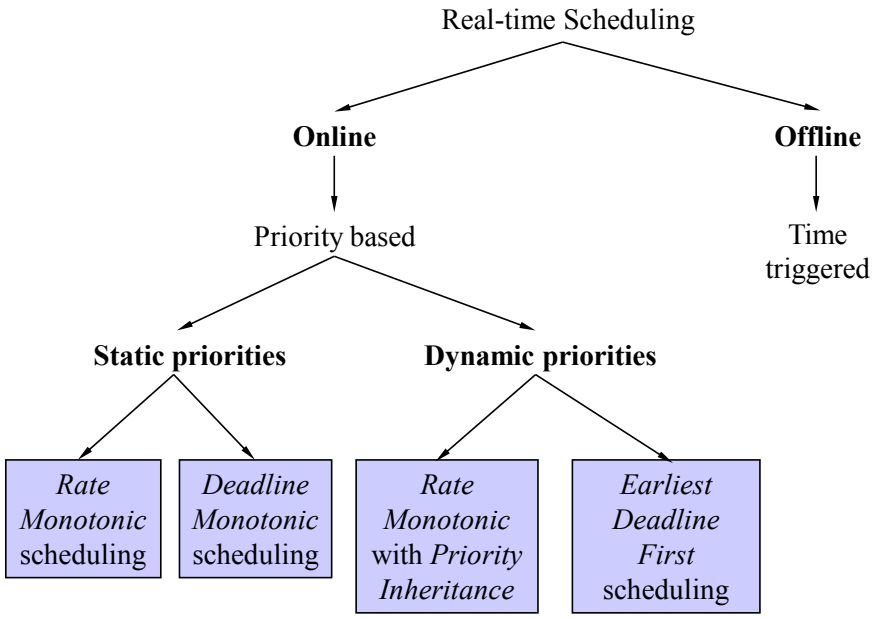


Figure 7: Classification of real-time scheduling

3.8 Rate Monotonic scheduling

The pioneering work on *Rate Monotonic* (RM) that was presented in early 70's (Liu & Layland, 1973), is of the oldest and most used scheduling methods. It is an online scheduling algorithm that uses static priority assignment for tasks. The static priorities are assigned on the basis of the task *period*: the shorter the period is, the higher is the task's priority. Since task periods are constant, priorities are assigned to tasks before execution and do not change over

time. The Rate Monotonic priority assignment is *optimal* meaning that if any static priority scheduling algorithm can meet all the deadlines, then the Rate Monotonic algorithm can too.

The original version of Rate Monotonic assumes that the deadline of each task is equal to its period. It also assumes that the tasks are independent, i.e., do not communicate to each other or share any resources. Moreover, Rate Monotonic is inherently preemptive; the currently executing task is immediately preempted by a newly arrived task with shorter period.

We will first present the original Rate Monotonic that operates according to those assumptions, and then we will extend it to allow for shared resources between tasks, which is more realistic to assume in system today.

Schedulability analysis of Rate Monotonic

As mentioned above, the configuration part of RM is to statically assign priorities to tasks based on their deadlines; the shorter the deadline, the higher priority. The run-time dispatching part is even simpler; the task with the highest priority in the ready queue gets to execute. We will now present the analysis part of Rate Monotonic.

Liu & Layland proved that for a set of n periodic tasks with unique periods, a feasible schedule that will always meet deadlines exists if the processor utilization is below a specific bound (depending on the number of tasks). The schedulability test for Rate Monotonic is given by:

$$U \leq n(2^{\frac{1}{n}} - 1) \tag{2}$$

The left part of the equation is the system utilization, and it will depend on the task periods and execution times, see equation (1). The right side is the *upper bound* for the schedulability and it will depend on the number of tasks in the task set. For example, the upper bound for two tasks is equal to $2(2^{1/2} - 1) = 0.828$, for three tasks is 0.78, and so on, see Figure 8.

Nr of tasks (n)	Upper bound
1	100 %
2	82.8 %
3	78.0 %
4	75.7 %
5	74.3 %
10	71.8 %
11	71.5 %
12	71.3 %
13	71.2 %
14	71.1 %
...	...
∞	69.3 %

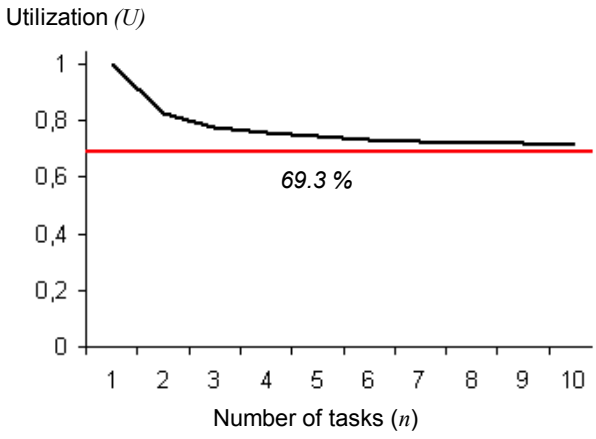


Figure 8: Upper bound for Rate Monotonic scheduling

When the number of tasks tends towards infinity this expression will tend towards:

$$\lim_{n\to\infty} n\left(2^{\frac{1}{n}}-1\right) = \ln 2 \approx 0.693\ldots$$

So, a rough estimate is that Rate Monotonic in the general case can meet all the deadlines if processor utilization is kept below 69.3%.

Example: Is the following task set schedulable by Rate Monotonic? If yes, show an execution trace of the tasks.

Task	C_i	T_i
τ_1	1	3
τ_2	1	6
τ_3	1	5

Task priorities:

Task τ_1 has the shortest period, 3, so it gets the highest priority. Task τ_2 has the longest period, 6, so it gets the lowest priority. Task τ_3 gets middle priority.

Utilization factor:

$$U = \frac{C_1}{T_1}+\frac{C_2}{T_2}+\frac{C_3}{T_3} = \frac{1}{3}+\frac{1}{6}+\frac{1}{5} = \frac{21}{30} = 0.7$$

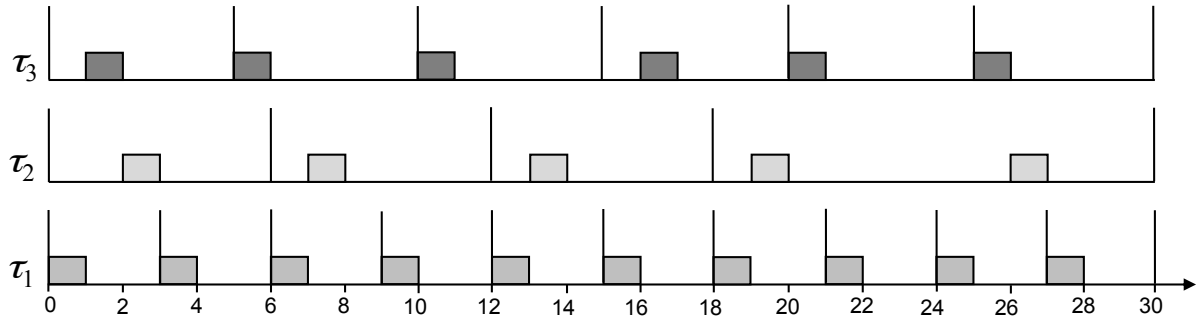
Schedulability test:

Is it true that $U \le n(2^{\frac{1}{n}}-1)$? $\quad n\left(2^{\frac{1}{n}}-1\right) = 3\left(2^{\frac{1}{3}}-1\right) = 0.78$

Yes, the utilization is less than 0.78, hence the task set is schedulable.

Execution trace:

Task τ_1 has the highest priority, hence it will start to execute its instances as soon as they become ready, i.e., at times 0,3,6,9, and so on. Task τ_3 has middle priority and its instances will be pre-empted by τ_1 whenever they are both ready to execute. Finally, task τ_2 has the lowest priority; hence it will be preempted both by τ_1 and τ_3.



The example above presents a case when the utilization is less or equal to the upper bound for the schedulability. We will now take a look at an example when the schedulability test is not fulfilled.

Example: Assume the same periodic task set as in the previous example, extended by an additional task τ1. Is the set schedulable by Rate Monotonic?

Task	C_i	T_i
τ_1	1	3
τ_2	1	6
τ_3	1	5
τ_4	2	10

Task priorities:

Task τ_4 gets the lowest priority, since it has the longest period. The remaining tasks have the same priority relative to each other as in the previous example.

Utilization factor:

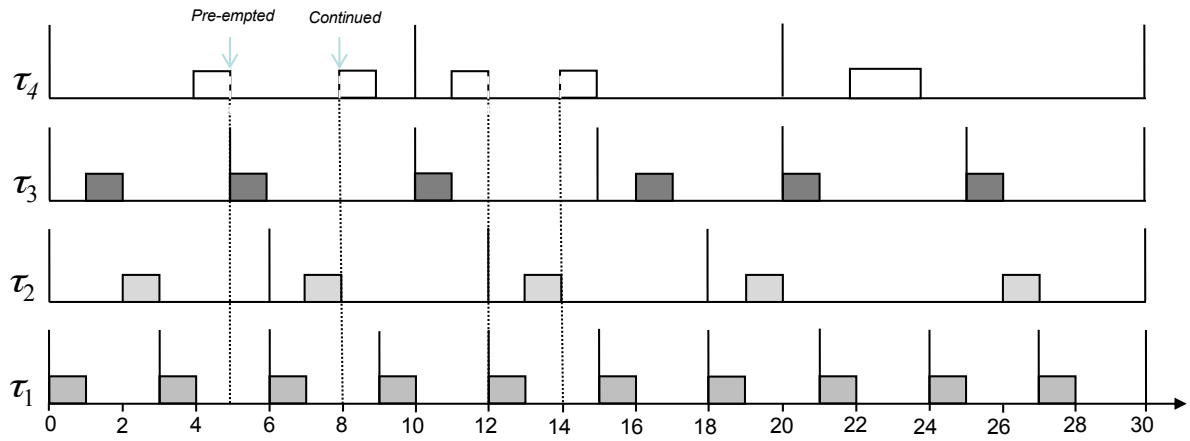
$$U = \frac{C_1}{T_1} + \frac{C_2}{T_2} + \frac{C_3}{T_3} + \frac{C_4}{T_4} = \frac{1}{3} + \frac{1}{6} + \frac{1}{5} + \frac{2}{10} = \frac{27}{30} = 0.9$$

Schedulability test:

Is it true that $U \leq n(2^{\frac{1}{n}} - 1)$? $\quad n\left(2^{\frac{1}{n}} - 1\right) = 4\left(2^{\frac{1}{4}} - 1\right) = 0.75$

No, the utilization is *not* less or equal than 0.75. We did not say anything about this case yet; we only said that if the schedulability test is fulfilled the task set is schedulable. Let's first look at the execution trace before making any conclusions.

Execution trace:



Task τ4 will be ready at time 0 but it will not be able to start executing before time 4 due to higher priority tasks. It will run for one clock tick (of its two required ticks) and then get pre-empted by τ3. Similarly, the second instance of τ4 will be ready at time 10 but it will start at time 11, get pre-empted at time 12, resume execution at time 14, and finish at time 15.

Finally, the last instance of τ4 in the hyper period will execute without pre-emptions between clock ticks 22 and 24. Hence, the task set is schedulable.

As we can see in the example above, a task set can be schedulable even if the schedulability test fails. This means that the schedulability test for Rate Monotonic, given in by the equation (2), is *sufficient* but *not necessary*. In other words, if the test is fulfilled, we can claim that the task set is schedulable, but if the test fails, i.e., $U > n(2^{1/n} - 1)$ and $U<1$, then the test cannot provide an unambiguous answer; maybe it is schedulable, maybe not – we simply cannot tell.

If the schedulability test for Rate Monotonic fails, we need an additional schedulability test, called *Response Time Analysis*, to make a conclusion whether a task set is schedulable or not. Since this test is not only used for Rate Monotonic, but even for other static priority assignment algorithms, we will first present an another algorithm in this class, *Deadline Monotonic*, and then explain Response Time Analysis, which is used in both methods.

3.9 Deadline Monotonic scheduling

According to Rate Monotonic, tasks with short periods will always be prioritized. However, this may not always be a good policy; there may exists tasks in the system that have long periods, but still they may be very important and hence, prioritized. For example, for sporadic tasks, the required response time is not, in general, related to the worst-case arrival rate (period) of the instances. Indeed, the characteristics of such tasks often demand a short response time compared to their minimum inter-arrival time. Hence, their deadlines are set to be considerably shorter than their periods, and Rate Monotonic, which assumes deadline is equal to period, cannot be used. Instead, *Deadline Monotonic* priority assignment can be used.

In Deadline Monotonic (DM), priorities assigned to tasks are inversely proportional to the length of the deadline. Thus, the shorter the relative deadline, the higher the priority. Deadline Monotonic algorithm was first proposed in 1982 (Leung & Whitehead, 1982) as an extension of Rate Monotonic where tasks can have relative deadlines less than their periods. Later, a schedulabilty analysis for this algorithm was proposed (Audsley, Burns, Richardson, & Wellings, 1991)

Note that Deadline Monotonic is a generalization of Rate Monotonic; this priority ordering defaults to a Rate Monotonic ordering when periods are equal to deadlines. Clearly, when the relative deadline of every task is equal to its period, both algorithms perform the same. When the relative deadlines are arbitrary, Deadline Monotonic performs better in a sense that it can sometimes produce a feasible schedule when Rate Monotonic fails.

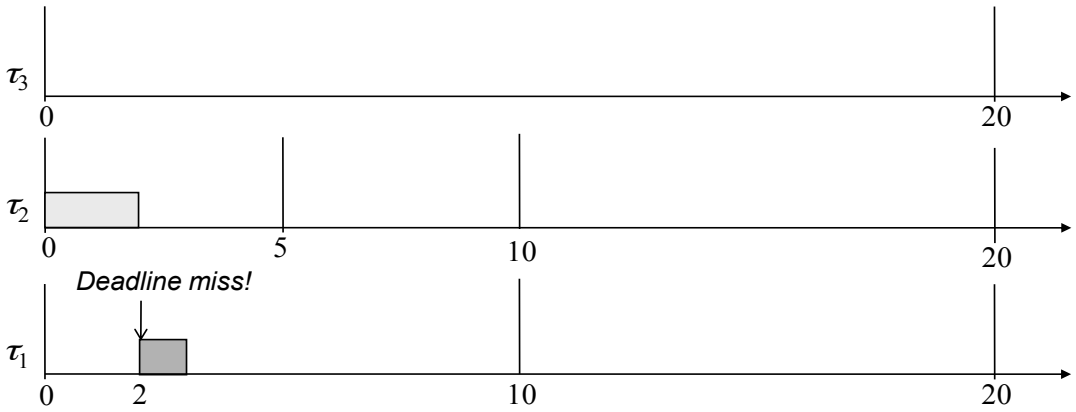
Example: Compare the schedulability of the following task set for Rate Monotonic and Deadline Monotonic, by drawing an execution trace.

Task	C_i	D_i	T_i
τ_1	1	2	10
τ_2	2	4	5
τ_3	4	10	20

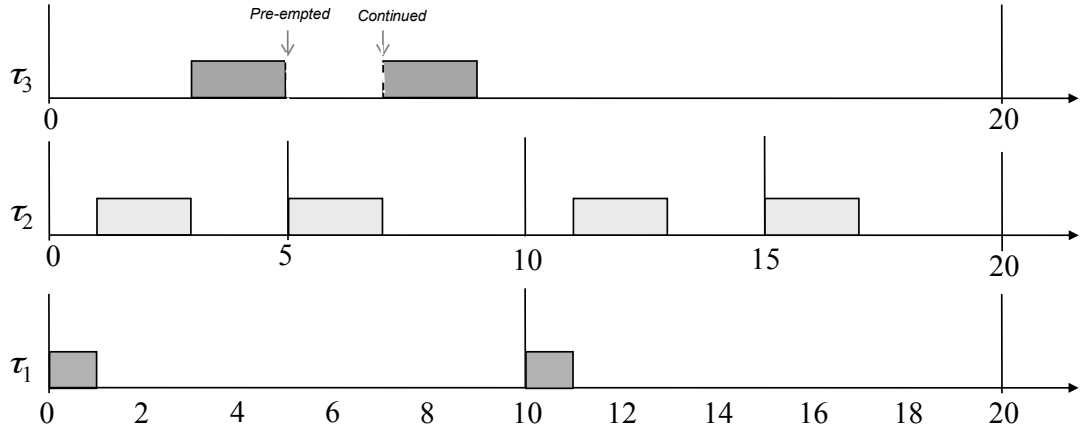
Priorities Rate Monotonic:
τ_2 = high, τ_1 = middle, τ_3 = low

Priorities Deadline Monotonic:
τ_1 = high, τ_2 = middle, τ_3 = low

Trace Rate Monotonic – deadline missed:



Trace Deadline Monotonic – schedulable:



Schedulability analysis of Deadline Monotonic

As mentioned before, for deadline equal to period, D = T, Rate Monotonic was proven to be optimal. Deadline Monotonic extends this optimality for D < T; if any static priority scheduling algorithm can schedule a set of tasks with deadlines unequal to their periods, then Deadline Monotonic will also schedule that task set, see (Leung & Whitehead, 1982) for the proof.

The schedulability of a task set with Deadline Monotonic could be guaranteed using the Rate Monotonic schedulability test, by reducing periods of the tasks to their relative deadlines:

$$\sum_{i=1}^{n} \frac{C_i}{D_i} \leq n(2^{\frac{1}{n}} - 1) \tag{3}$$

However, such a test would not be optimal as the workload on the processor would be overestimated. Besides, the test above is sufficient but not necessary. Hence, we need another schedulability test, just like for Rate Monotonic when the utilization test does not provide an answer. This new test is called *Response Time Analysis*, and it will described next.

3.10 Response Time Analysis

Utilization-based tests have two significant drawbacks: they are not exact, and they are not really applicable to a more general task model. We could see that the utilization test for Rate Monotonic, presented in equation (2), is not necessary, i.e., if the utilization is greater that the upper bound we could now draw any conclusions about the task set schedulability. It is even worse for Deadline Monotonic, see equation (3), because the test will assume that the task instances will be activated more often than they actually are (the instances are activated at the beginning of each new period, but the test uses deadlines, which are shorter than periods). Besides, utilization-based tests only provide a yes/no answer. They do not give any indication of actual response times of the tasks.

Response Time Analysis (Joseph & Pandya, 1986) overcomes these drawbacks. It calculates the response times of the tasks and compares it to their deadlines. A task τ_i is said to be schedulable if:

$$R_i \leq D_i \tag{4}$$

Remember that R_i is the worst-case response time of task τ_i , i.e., the longest response time of all task instances. Hence, if the task instance with the worst-case response time will finish before its deadline, then all other instances of the same task will also do. We will refer to this worst-case response time just as response time in the remainder of the text.

For a set of tasks, the set is schedulable if response times of *all* tasks in the set are less or equal than their assigned deadlines. The test is both sufficient and necessary for Rate Monotonic and Deadline Monotonic algorithms. Actually, Response Time Analysis can be used even for other static priority algorithms; as long as the priorities are set before run-time and do not change over time, the test will provide an answer. We will now see how response times for tasks are calculated.

When calculating the response time of a task τ_i, the worst case will occur when all higher priority tasks are invoking their instances at the same time. The basic idea of Response Time Analysis is to find an equation that will calculate the response time R_i, assuming pre-emptions from high priority instances. Hence, R_i is made up of two times: the time task τ_i takes to execute its own code, C_i, and the time it takes for higher priority tasks to execute:

$$R_i = C_i + I_i$$

The term I_i, called *interference*, is the pre-emption time from higher priority tasks. For the highest priority task in the system, its response time will be equal to its own execution time, $R_i=C_i$, since no other tasks will preempt. Other tasks will suffer interference from higher priority tasks. The problem now becomes finding the interference time for a task τ_i.

Let τ_k be a task with higher priority than τ_i, that is released at the same time as τ_i. i.e., at time t. Task τ_i will have an instance that becomes ready at t and finishes its execution at time $t+R_i$ (we assume this is the instance with the worst-case response time). During its execution, the instance of τ_i will be pre-empted by the higher priority task τ_k. If τ_k has a shorter period than τ_i then it will pre-empt τ_i several times, i.e., several instances of τ_k will occur and pre-empt the current instance of τ_i.

The number of instances of τ_k that occur in the interval $[t,\ t+R_i]$, and hence interfere (pre-empt) τ_i can be calculated by dividing the length of the interference interval by the activation frequency of τ_k, i.e. its period time:

$$\left\lceil \frac{R_i}{T_k} \right\rceil$$

The symbol ⌈ ⌉ is the ceiling function, and is a round-up function. So, for example$\lceil 2.3 \rceil = 3$.

The total time taken by task τ_k when it pre-empts and executes is simply the number of instances of τ_k, calculated as above, multiplied by its execution time, C_k:

$$\left\lceil \frac{R_i}{T_k} \right\rceil C_k$$

So, for the total interference term, we simply add this up for all the higher priority tasks than τ_i:

$$I_i = \sum_{\forall k \in hp(\tau_i)} \left\lceil \frac{R_i}{T_k} \right\rceil C_k$$

Where $hp(\tau_i)$ is the set of tasks that have higher priority than task τ_i .

Hence, the worst-case response time for a task τ_i is given by:

$$R_i = C_i + \sum_{\forall k \in hp(\tau_i)} \left\lceil \frac{R_i}{T_k} \right\rceil C_k$$

Since R_i is represented on both sides of the equation, it can be solved by forming a recurrence relation, i.e., the next value is obtained based on the currently calculated value:

$$R_i^{n+1} = C_i + \sum_{\forall k \in hp(\tau_i)} \left\lceil \frac{R_i^n}{T_k} \right\rceil C_k \tag{5}$$

The initial value of for R_i is the execution time C_i, since the shortest possible response time of τ_i is its worst-case execution time. The set of consecutive values $\{R_0, R_1, R_2, ...\}$ is monotonically non-decreasing, i.e., the next value is greater of equal to the previously calculated value, and the sequence will converge to the smallest value of R_i that satisfies equation (5).

Simply said, we stop iterating when $R_i^{n+1} = R_i^n$, and if $R_i \leq D_i$, we conclude that the task is schedulable. Otherwise, if some of the iteration steps result in the value of R_i that is larger than D_i, we should stop with the iterations and conclude that the task is not schedulable.

Example: Is the following task set schedulable by Rate Monotonic?

Task	C_i	T_i=D_i
τ_1	1	3
τ_2	1	6
τ_3	1	5
τ_4	2	10

An observant reader will notice that this task set is the same as one presented in subsection 3.8, where we explained Rate Monotonic algorithm. We calculated before that $U=0.9$ is larger than the upper bound 0.75. Hence, the utilization test could not give us an answer whether the task set is schedulable or not. It was first after running and analyzing the execution trace that we could see that the set was schedulable.

Instead of analyzing the execution trace, we can use Response Time Analysis to come to the same conclusion *before run-time*. We use the equation (5) to calculate response times for all tasks and compare then to their deadlines.

Response time of task τ_1:

According to Rate Monotonic, task τ_1 is assigned the highest priority. The set of high priority tasks $hp(\tau_1)$ is empty, hence the response time of τ_1 will be equal to its execution time:

$$hp(\tau_1) = \{\ \} \Rightarrow R_1 = C_1 = 1 \leq D_1$$

Response time of task τ_2:

Task τ_2 will be assigned higher priority than τ_4, but lower than τ_1 and τ_3, which both have shorter periods. Hence, both τ_1 and τ_3 will influence the response time of τ_2.

$hp(\tau_2) = \{\tau_1, \tau_3\}$

$$R_2^0 = C_2 = 1$$

$$R_2^1 = C_2 + \left\lceil \frac{R_2^0}{T_1} \right\rceil C_1 + \left\lceil \frac{R_2^0}{T_3} \right\rceil C_3 = 1 + \left\lceil \frac{1}{3} \right\rceil 1 + \left\lceil \frac{1}{5} \right\rceil 1 = 1 + 1 + 1 = 3$$

$$R_2^2 = C_2 + \left\lceil \frac{R_2^1}{T_1} \right\rceil C_1 + \left\lceil \frac{R_2^1}{T_3} \right\rceil C_3 = 1 + \left\lceil \frac{3}{3} \right\rceil 1 + \left\lceil \frac{3}{5} \right\rceil 1 = 1 + 1 + 1 = 3$$

Since $R_2^2 = R_2^1, we\ can\ stop\ iterations. Hence\ R_2 = 3 \leq D_2$

Response time of task τ_3:

$hp(\tau_3) = \{\tau_1\}$

$$R_3^0 = C_3 = 1$$

$$R_3^1 = C_3 + \left\lceil \frac{R_3^0}{T_1} \right\rceil C_1 = 1 + \left\lceil \frac{1}{3} \right\rceil 1 = 1 + 1 = 2$$

$$R_3^2 = C_3 + \left\lceil \frac{R_3^1}{T_1} \right\rceil C_1 = 1 + \left\lceil \frac{2}{3} \right\rceil 1 = 1 + 1 = 2$$

$$R_3^2 = R_3^1 \Rightarrow R_3 = 2 \leq D_3$$

Response time of task τ_4:

$hp(\tau_4) = \{\tau_1, \tau_2, \tau_3\}$

$$R_4^0 = C_4 = 2$$

$$R_4^1 = C_4 + \left\lceil \frac{R_4^0}{T_1} \right\rceil C_1 + \left\lceil \frac{R_4^0}{T_2} \right\rceil C_2 + \left\lceil \frac{R_4^0}{T_3} \right\rceil C_3 = 2 + \left\lceil \frac{2}{3} \right\rceil 1 + \left\lceil \frac{2}{6} \right\rceil 1 + \left\lceil \frac{1}{5} \right\rceil 1 = 5$$

$$R_4^2 = C_4 + \left\lceil \frac{R_4^1}{T_1} \right\rceil C_1 + \left\lceil \frac{R_4^1}{T_2} \right\rceil C_2 + \left\lceil \frac{R_4^1}{T_3} \right\rceil C_3 = 2 + \left\lceil \frac{5}{3} \right\rceil 1 + \left\lceil \frac{5}{6} \right\rceil 1 + \left\lceil \frac{5}{5} \right\rceil 1 = 6$$

$$R_4^3 = C_4 + \left\lceil \frac{R_4^2}{T_1} \right\rceil C_1 + \left\lceil \frac{R_4^3}{T_2} \right\rceil C_2 + \left\lceil \frac{R_4^3}{T_3} \right\rceil C_3 = 2 + \left\lceil \frac{6}{3} \right\rceil 1 + \left\lceil \frac{6}{6} \right\rceil 1 + \left\lceil \frac{6}{5} \right\rceil 1 = 7$$

$$R_4^4 = C_4 + \left\lceil \frac{R_4^3}{T_1} \right\rceil C_1 + \left\lceil \frac{R_4^3}{T_2} \right\rceil C_2 + \left\lceil \frac{R_4^3}{T_3} \right\rceil C_3 = 2 + \left\lceil \frac{7}{3} \right\rceil 1 + \left\lceil \frac{7}{6} \right\rceil 1 + \left\lceil \frac{7}{5} \right\rceil 1 = 9$$

$$R_4^5 = C_4 + \left\lceil \frac{R_4^4}{T_1} \right\rceil C_1 + \left\lceil \frac{R_4^4}{T_2} \right\rceil C_2 + \left\lceil \frac{R_4^4}{T_3} \right\rceil C_3 = 2 + \left\lceil \frac{9}{3} \right\rceil 1 + \left\lceil \frac{9}{6} \right\rceil 1 + \left\lceil \frac{9}{5} \right\rceil 1 = 9$$

$$R_4^5 = R_4^4 \Rightarrow R_4 = 9 \leq D_4$$

We see that all response times are less than deadlines; hence the task set is schedulable by Rate Monotonic.

If we compare all calculated response times in the example above with the execution trace response times of the example in subsection 3.8, we will see that they are the same (for the worst case scenario when all tasks are released at the same time). We can say that Response Time Analysis simulates (before run-time) the actual execution pattern that will occur at run-time. This way, we can analyze a real-time system for its schedulability before the system starts to operate.

3.11 Static priority scheduling with shared resources

So far, we presented some scheduling algorithms which assume independent tasks, i.e., the tasks were assumed not to share any system resources. However, this is not realistic to assume in many real-time systems where tasks usually share resources.

Resource sharing between concurrent tasks introduces new challenges to be solved. For example, the concurrent update problem is well known to programmers of concurrent systems, where tasks write over each other values. Moreover, high priority tasks can get blocked waiting for a resource that is currently used by a low priority task.

A number of solutions have been proposed. The most common of these is the semaphore (invented in the 1960's by Djikstra), see Chapter 2. The basic idea is that a task can only access shared data after it has requested and locked a semaphore. When the task has finished with the resource, it unlocks the semaphore.

We will start by introducing the problems when tasks share resources, e.g., blocking, priority inversion, deadlock, etc. Then, we will present some real-time resource access protocols, such as Priority Inheritance Protocol and Priority Ceiling Protocol. Finally, we will extend the schedulability analysis of static priority scheduling to include shared resources.

Blocking

A task is said to be *blocked* (i.e., transferred into the blocked state) when it has been released, or it has been executing for a while, but it cannot proceed the execution due to shared

resource that is currently used by some other, lower priority task; the higher priority task is blocked until the resource becomes free. This situation is illustrated in Figure 9.

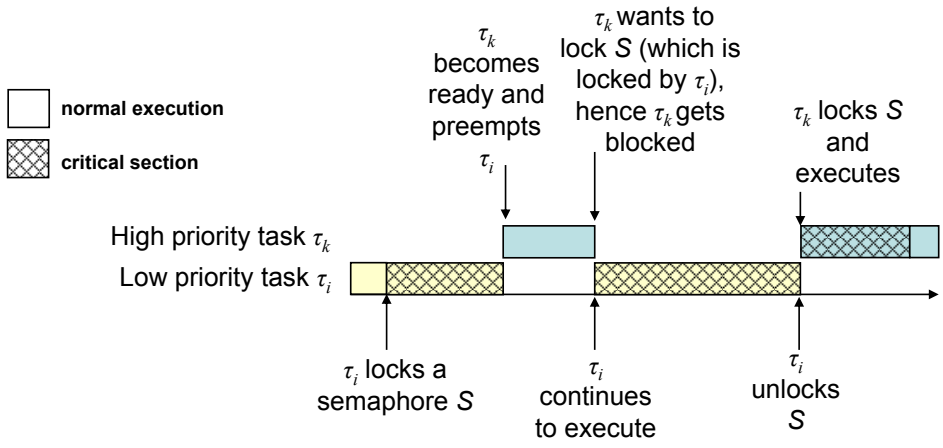


Figure 9: Blocking

Blocking has a negative effect on schedulability and predictability; it delays the execution of high priority tasks. The response time analysis that we previously presented does not takes into the account the effects of blocking, hence it has to be modified. But before we do that, we will have a look at some other problems caused by blocking, and the solutions in form of real-time resource reservation protocols.

Priority inversion problem

Priority inversion is a phenomenon for which a task is blocked by lower priority tasks for an *unbounded* amount of time. This can be best explained with an example. Assume the same two tasks as in the previous example with blocking, extended with an additional, middle priority task τ_j i.e., task τ_j has higher priority than τ_i but lower priority than τ_k. Furthermore, assume that τ_j does not want to use the same resource that is shared between τ_i and τ_k.

Assume now that τ_j becomes ready while τ_k is blocked by τ_i. Then, τ_j will preempt τ_i and execute without interruptions, since it does not request the resource currently held by τ_i. When τ_j is done, τ_i will resume its execution, execute, and then release the semaphore, so that τ_k can use it.

The whole situation is depicted in Figure 10. We see that task τ_k is delayed not only by τ_i who holds a resource that τ_k needs, but also it is *indirectly* delayed by τ_j who does not even use the same resource. Now imagine several such tasks with priority values between τ_i's and τ_k's priorities that are activated while τ_k is blocked by τ_i – *each* of them will delay the execution of τ_k although they have lower priorities than τ_k. This means that the response time of τ_k can be very long (unbounded), despite the fact it has the highest priority.

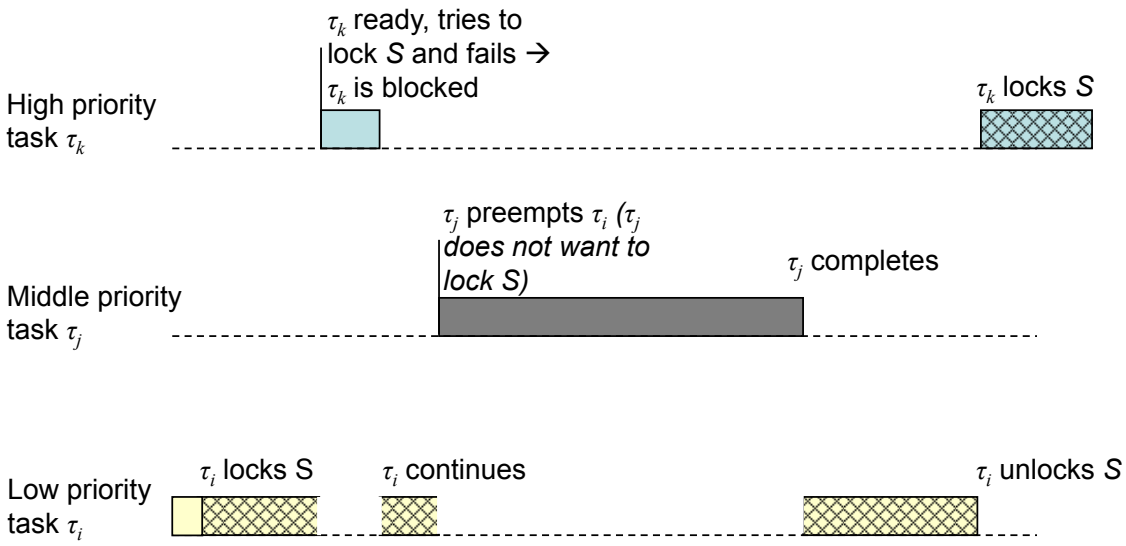


Figure 10: Priority inversion problem

We mentioned in Chapter 2 that general-purpose operating systems, such as Windows, do not prevent this phenomenon, and hence they can be dangerous to use in hard real-time systems. We will now see how priority inversion can be avoided.

Priority Inheritance Protocol

Priority Inheritance Protocol (PIP) is a policy for resource usage that prevents priority inversion. The main idea is to temporarily swap the priorities between a high priority task and a low priority task that is currently blocking it, in order to prevent any middle priority tasks to execute in between. When the high priority task gets blocked, the low priority task *inherits* the priority from the high priority task. When the low priority task is done with the semaphore, it resumes the priority it had at the point when the high priority task got blocked.

Figure 11 shows how PIP solves the priority inversion problem of the previous example. When task τ_k gets blocked by τ_i, then τ_i inherits the priority of τ_k, i.e., it continues to execute with high priority. This means that task τ_j cannot preempt it, since τ_i has (temporarily) higher priority than τ_j. Hence, τ_j will not be able to execute and delay τ_k, as it did in the previous example. We can say that τ_j is blocked by τ_i who inherited higher prority. This type of blocking is known as *push-through* blocking.

Neither any other potential tasks with the priority in between τ_i is τ_k will be able to preempt, hence the delay of the high priority task will be *minimized* to the length of the critical section of the low priority task that blocks it.

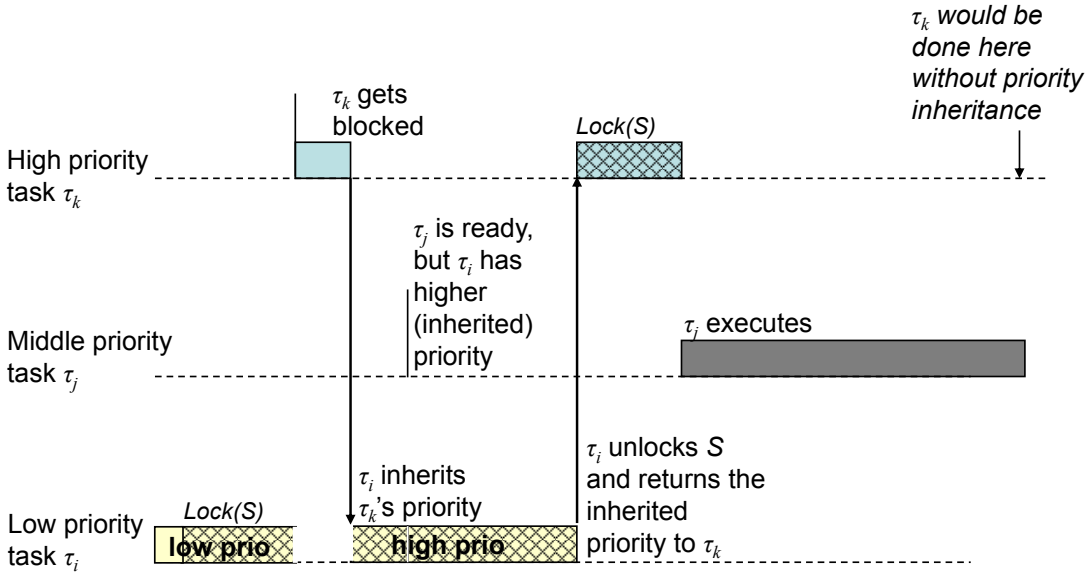


Figure 11: Priority Inheritance Protocol

Priority Inheritance Protocol prevents priority inversion, but it does not prevent deadlocks in the system (see chapter 2 for explanation of deadlock). Furthermore, it does not prevent chained blocking, i.e., a task that uses n semaphores can in the worst case be blocked *n* times by lower priority tasks holding the semaphores, see Figure 12.

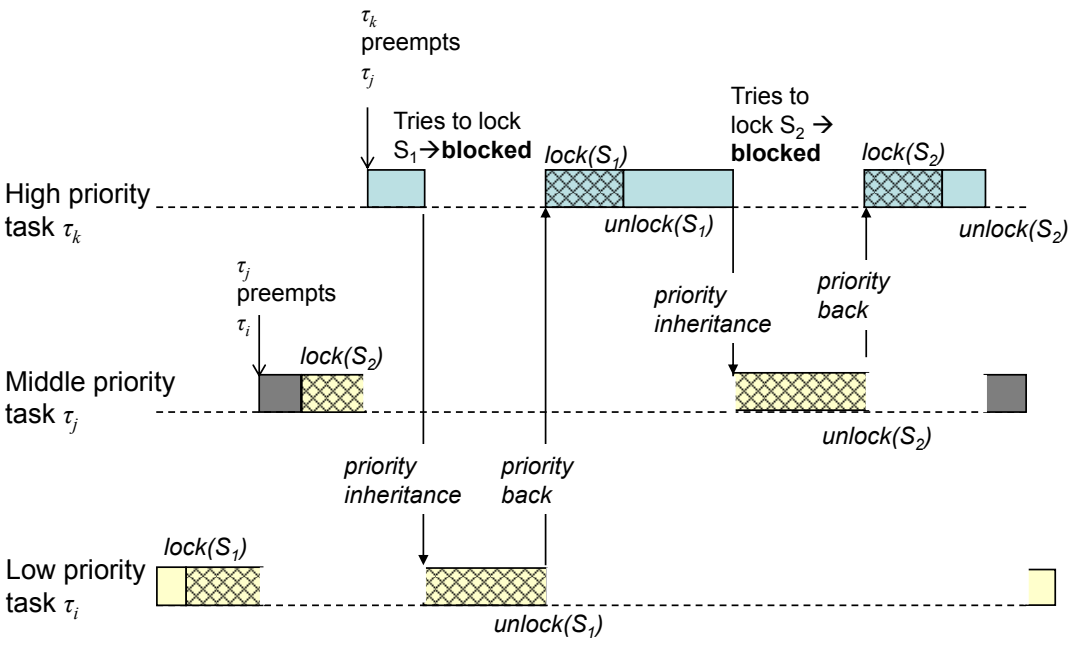


Figure 12: Chained blocking

We see in the figure that the highest priority task who wants to use two semaphores gets blocked twice before it completes. Luckily, there is a solution to both deadlocks and chained blocking, which we present next.

Priority Ceiling Protocol

Priority Ceiling Protocol (PCP) is an extension of Priority Inheritance Protocol, where each semaphore *S* has a statically assigned priority ceiling value, *ceil(S),* defined as the maximum priority of all tasks that may use the semaphore at run-time. The priority ceiling value is then used at run-time to determine if a task is allowed to lock *S* or not.

When a task τ_i requests a free semaphore S_k during its execution, the request is granted only if the priority of τ_i is strictly greater than the ceiling values of all semaphores currently allocated to other tasks, or if there are no other locked semaphores at the moment.

Here is an example. Assume three periodic tasks τ_i, τ_j and τ_k, with priorities assigned as in previous examples, i.e., τ_i has the lowest and τ_k the highest priority. There are also three semaphores in the system, S_1, S_2 and S_3 that are protecting some resources shared by the tasks. Assume that task τ_k wants to use only semaphore S_1, while tasks τ_i and τ_j are both using the other two semaphores, S_2 and S_3. The semaphores are requested by the tasks as follows:

Task τ_i (low priority=1)

```
while(1) {
    ...
    lock(S3);
    ...
    lock(S2);
    ...
    unlock(S2);
    ...
    unlock(S3);
}
```

Task τ_j (middle priority=2)

```
while(1) {
    ...
    lock(S2);
    ...
    lock(S3);
    ...
    unlock(S3);
    ...
    unlock(S2);
}
```

Task τ_k (high priority=3)

```
while(1) {
    ...
    lock(S1);
    ...
    unlock(S1);
}
```

Hence, the priority ceiling of the semaphores will be:

$$ceil(S_1) = priority(\tau_k) = 3\ (high)$$
$$ceil(S_2) = MAX\left(priority(\tau_i); priority(\tau_j)\right) = MAX(1;2) = 2\ (middle)$$
$$ceil(S_3) = MAX\left(priority(\tau_i); priority(\tau_j)\right) = MAX(1,2) = 2\ (middle)$$

Note that the ceilings are assigned before run-time. A possible run-time scenario, where the lowest priority task τ_i is released first (the other two are not ready yet) is showed in Figure 13.

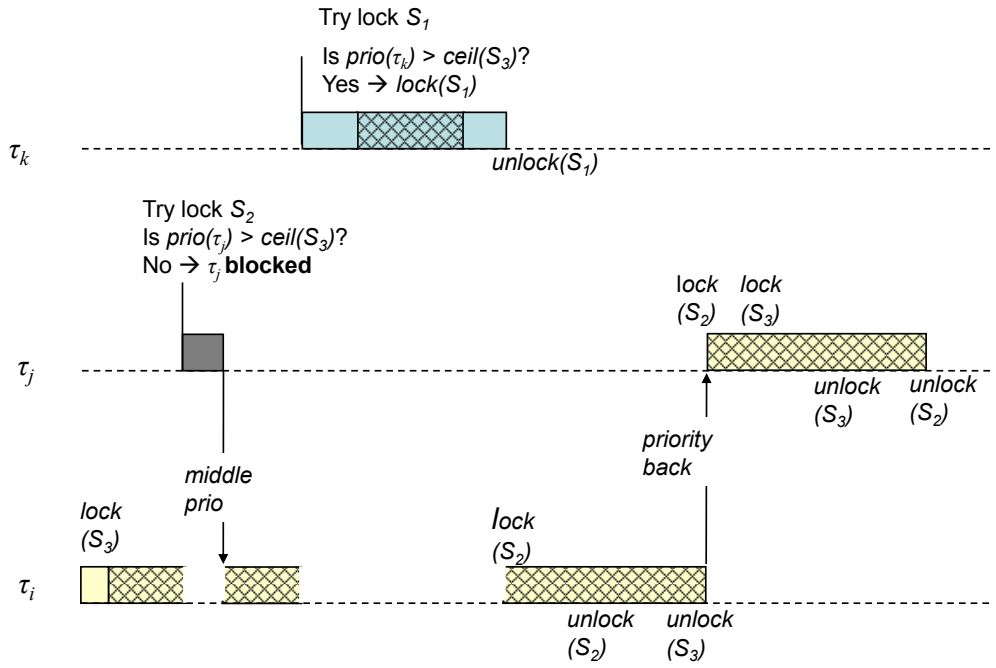


Figure 13: Priority Ceiling Protocol

This is what happens in the example: at the beginning, only tasks τ_i is ready. It starts to execute and eventually it tries to lock S_3. Since there are no locked semaphores at the moment, τ_i will be able to lock S_3. When τ_j becomes ready, it will preempt τ_i and, after a while, it will request semaphore S_2. This time, there are currently locked semaphores in the systems, namely semaphore S_3 that is locked by τ_i, which implies that τ_j must compare its own priority against the ceiling of S_3. Since the priority of τ_j is 2, and the ceiling of S_3 is also 2, the priority of τ_j is not strictly greater than the ceiling of S_3, hence τ_j will not be granted the requested semaphore S_2 and it becomes blocked. Task τ_i resumes its execution and after a while it gets preempted by τ_k. During its execution, τ_k will try to lock S_1, and since its priority is greater than the ceiling of all currently locked semaphores, i.e., semaphore S_3, it will be granted to lock S_1. The rest of the figure is quite self-explanatory.

Before introducing PCP, we had two types of blocking: *direct blocking*, which occurs when a higher priority task tries to acquire a resource already held by a lower priority task, and *push-through* blocking, which occurs when a middle priority task is blocked by a lower priority task that has inherited a higher priority form a task it directly blocks. Notice that PCP introduces a third form of blocking, called *ceiling blocking*, where tasks are blocked on free semaphores, e.g., task τ_2 is blocked on S_2 which is free when τ_2 requests it. This is necessary for avoiding deadlock and chained blockings.

Some good properties of PCP are: under PCP, a task instance can be blocked for at most the duration of one critical section. Furthermore, PCP automatically prevents deadlocks, since it induces an ordering on the way that resources can be requested. Also, from the implementation point of view, the major implications of PCP in the kernel data structures is that semaphore queues are not longer needed, since the tasks blocked by the protocol can be kept in the ready queue.

Immediate Ceiling Priority Protocol

Priority Ceiling Protocol introduces an extra implementation challenge: the scheduling must keep track of which task is blocked on which semaphore, and what the inherited priorities are. Furthermore, it introduces extra overhead to the scheduler to have to work out whether a task can lock a semaphore or not.

It turns out that there is a simple protocol, based on PCP, which has the same worst-case timing behavior. It's generally called the *Immediate Ceiling Priority Protocol*, ICPP (also known as Immediate Inheritance Protocol, IIP). It has the same model as the priority ceiling protocol (i.e., pyramid locking patterns, no holding of semaphores between instances, etc), but it has a different run-time behavior: when a task τ_i locks a semaphore S_k, the task *immediately* sets its own priority to the maximum of its current priority and the ceiling priority of S_k. When the task finishes with S_k, it sets its priority back to what it was before. Note the difference from Priority Ceiling Protocol: in PCP the priority is raised first when a higher priority task wants the same resource.

It is easy to see why a task τ_i is only delayed at most once by a lower priority task (just like with the PCP): there cannot have been two lower priority tasks that locked two semaphores with ceilings higher than the priority of task τ_i. This is because one of them will have instantly inherited a higher priority first. Because it inherits a higher priority, the other task cannot then run and lock a second semaphore.

In ICPP, it turns out that we do not actually need to lock or unlock S_k. The reason is simple: the semaphore S_k cannot have been locked when task τ_i comes to lock it, because otherwise there would be another task running with the same priority as τ_i, and task τ_i would not be executing. If task τ_i was not executing, then it could not execute code to try lock S_k in the first place.

Because the inheritance is immediate, task τ_i is blocked, if at all, before it starts running. This is because, if a lower priority task holds a semaphore that can block task τ_i it will be running at a priority at least as high as task τ_i. Hence, when task τ_i is invoked it will not start running until the lower priority task has finished.

In ICPP, once a task starts it cannot be blocked, though its start may be delayed by lower-priority tasks locking resources it may use. It is less complex than PCP, and it has fewer context switches. On the other hand, PCP gives better concurrency, since tasks are blocked later than in ICPP, i.e., when they attempts to lock a resource, not at the start of the execution. It also avoids unnecessary delay of tasks that do not lock resources. What is common for both protocols is that worst case blocking time is single largest critical region of any lower priority task that accesses resources with ceilings at or above task's priority. Because the worst-case timing performance of ICPP protocol is the same as PCP, the analysis developed for PCP (i.e., the calculation of blocking factors) remains unchanged for ICPP.

Response Time Analysis with blocking

We will now see how we can extend the response time analysis in order to include effects of blocking. As we showed above, a task τ_i can be blocked by lower priority tasks due to shared resources. The delay caused by blocking is called *blocking factor*, and for task τ_i is denoted as B_i. It is a function of the length of critical sections of the lower priority tasks that can block. In other words, the blocking factor of a task τ_i is the longest time a task can be delayed by the execution of lower priority tasks.

In PCP, a given task τ_i is blocked by at most *one* critical section of any lower priority task, among all tasks that can lock a semaphore with priority ceiling greater than or equal to the priority of task τ_i:

$$B_i = MAX\left(cs(\tau_j, S_k) \mid \tau_j \in lp(\tau_i) \wedge \exists S_k, priority(\tau_i) \leq ceil(S_k)\right)$$

where $cs(\tau_i)$ is the length of the critical section of task τ_i, and $lp(\tau_i)$ is the set of lower priority tasks than τ_i.

What this means is: first we identify all the tasks with lower priority than task τ_i. Second, we identify all the semaphores that the lower priority tasks can lock. Third, we select from those only the semaphores where the ceiling of the semaphore has a priority higher than or the same as the priority of task τ_i. Finally, we look at the computation time that each lower priority tasks is holding its semaphores (critical section), and the longest of these computation times is the blocking factor, B_i.

Once when we calculated the blocking factor, it is easily added in to the response time analysis equation:

$$R_i^{n+1} = C_i + \boldsymbol{B_i} + \sum_{\forall k \in hp(\tau_i)} \left\lceil \frac{R_i^n}{T_k} \right\rceil C_k \qquad (6)$$

Example: Is the following task set schedulable by Rate Monotonic if Priority Ceiling Protocol is used for shared resources?

τ_i	C_i	T_i=D_i	S_k	$cs(\tau_i,S_k)$
τ_1	10	100	S_1	*1*
τ_2	12	40	S_1	*2*
			S_2	*1*
τ_3	6	50	S_1	*1*

Task priorities:
priority(τ_1) = 1 (low)
priority(τ_2) = 3 (high)
priority(τ_3) = 2 (middle)

Semaphore priority ceilings:
ceil(S_1) = MAX(*prio*(τ_1); *prio*(τ_2); *prio*(τ_3)) = 3
ceil(S_2) = *prio*(τ_2) = 3

Blocking factors:

Task τ_1:

A task gets blocked because of the *lower priority* tasks that lock semaphores. Since τ_1 has the lowest priority in the system, its blocking time is equal to zero, i.e.:

$B_1 = 0$

Task τ_2:

Lower priority tasks: $lp(\tau_2) = \{\tau_1, \tau_3\}$

Tasks τ_1 and τ_3 are using semaphore S_1, hence the blocking factor will be the maximum critical section of two of them:

$B_2 = \text{MAX}(\ cs(\tau_1, S_1)\ ;\ cs(\tau_3, S_1)\ ) = \text{MAX}(1;1) = 1$

Task τ_3:

Lower priority tasks: $lp(\tau_3) = \{\tau_1\}$

Task τ_1 is using semaphore S_1, hence the blocking factor will be:

$B_3 = cs(\tau_1, S_1) = 1$

Response times:

Task τ_1:

$$hp(\tau_1) = \{\tau_2, \tau_3\}$$

$$R_1^0 = C_1 = 10$$

$$R_1^1 = C_1 + B_1 + \left\lceil \frac{R_1^0}{T_2} \right\rceil C_2 + \left\lceil \frac{R_1^0}{T_3} \right\rceil C_3 = 10 + 0 + \left\lceil \frac{10}{40} \right\rceil 12 + \left\lceil \frac{10}{50} \right\rceil 6 = 28$$

$$R_1^2 = 10 + 0 + \left\lceil \frac{28}{40} \right\rceil 12 + \left\lceil \frac{28}{50} \right\rceil 6 = 28 \Rightarrow R_i = 28 < D_1$$

Task τ_2:

$$hp(\tau_2) = \{\ \ \} \Rightarrow R_2 = C_2 + B_2 = 12 + 1 = 13 < D_2$$

Task τ_3:

$$hp(\tau_3) = \{\tau_2\}$$

$$R_3^0 = C_3 = 6$$

$$R_3^1 = C_3 + B_3 + \left\lceil \frac{R_3^0}{T_2} \right\rceil C_2 = 6 + 1 + \left\lceil \frac{6}{40} \right\rceil 12 = 19$$

$$R_3^2 = 6 + 1 + \left\lceil \frac{19}{40} \right\rceil 12 = 19 \Rightarrow R_3 = 19 < D_3$$

Answer: The task set is schedulable.

This concludes the discussion on static priority scheduling algorithms. Now we will present an algorithm for online scheduling that is based on *dynamically* assigned task priorities.

3.12 Earliest Deadline First

Earliest Deadline First (EDF) algorithm is a *dynamic priority* assignment algorithm that schedules tasks according to their absolute deadlines; task instances with earlier deadlines are given higher priorities and scheduled first. The early idea of EDF algorithm was first proposed by Jackson, already in 1955, and later formally proven for optimality by Derouzos in 1974. The upper bound for the utilization has been calculated by Liu and Layland, in 1973, and published in the same article as their work on Rate Monotonic (Liu & Layland, 1973).

EDF has been proven to be optimal scheduling policy in the sense of feasibility. This means if there exists a feasible schedule for a task set, then EDF is able to find it. Notice that EDF does not make any specific assumption on the periodicity of the tasks. Hence, it can be used for scheduling periodic as well as aperiodic tasks.

The absolute deadline of a periodic task τ_i depends on the current j^{th} instance as:

$$d_i^j = (j-1)T_i + D_i$$

Hence, EDF is a dynamic priority assignment. Moreover, the currently executing task is preempted whenever another task instance (of another task) with earlier deadline becomes active. Consequently, at any point in time the task with the shortest time left until its deadline executes.

Similar to RM analysis, EDF analysis can also be divided into two cases: deadline equal to period, and deadline less than period. In the first case, the processor utilization analysis can be used to determine schedulablity under EDF. A set of periodic tasks is schedulable with EDF if and only if the total processor utilization is less than 100%:

$$U \leq 1 \tag{7}$$

So, it is enough that the total utilization is less than one to be able to claim the set is schedulable by EDF (compare it to Rate Monotonic, where $U \leq n(2^{1/n} - 1)$ must hold). Obviously, we achieve better processor utilization by using EDF.

Example: Is the following task set schedulable by EDF? If yes, draw an execution trace.

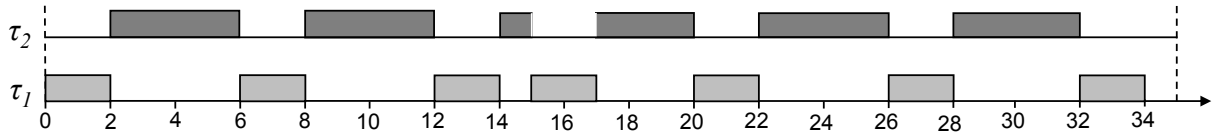
Task	C_i	T_i=D_i
τ_1	2	5
τ_2	4	7

Utilization factor:

$$U = \frac{C_1}{T_1} + \frac{C_2}{T_2} = \frac{2}{5} + \frac{4}{7} = \frac{34}{35} = 0.97$$

Yes, the task set is schedulable!

Execution trace:



At time 0, both tasks are ready, but the instance of task τ_1 has a shorter deadline and it will be scheduled first. At time 5, task τ_1 becomes ready again, but now it is the instance of task τ_2 that has shorter deadline, 7, than the instance of task τ_1 with the deadline of 10. Hence, task τ_2 will be given advance. The same occurs at time 10.

Notice that Rate Monotonic, who builds on static priorities, would schedule τ_1 at time 5, and hence task τ_2 would miss its deadline.

As we can see in the example above, EDF will always set highest priority to the task which current instance has shortest deadline. As a consequence, different task instances from the same task might have different priority, depending on the other ready tasks at the moment. Therefore we say that priority assignment is dynamic. On the contrary, in Rate Monotonic, all instances of the same task will have same priority and will not be changed under run-time (unless we use some of the semaphore protocols explained above.

Processor Demand Analysis

Processor utilization analysis for EDF is valid only if deadline is equal to period for all tasks in a task set. If this is not the case, i.e., deadline is less than period, *Processor Demand Analysis* (PDA) can be used.

The processor demand for a task τ_i in a given time interval [0, L] is the amount of processor time that the task needs in the interval in order to meet the deadlines that fall within the interval. Let N_i represent the number of instances of τ_i that must complete execution before L. We can calculate N_i by counting how many times task τ_i has been released during the interval [0, L-D_i], i.e., N_i can be expressed as:

$$N_i = \left\lfloor \frac{L - D_i}{T_i} \right\rfloor + 1$$

The total processor demand for all tasks in the interval is thus:

$$C^T(0,L) = \sum_{i=1}^{n} N_i\, C_i = \sum_{i=1}^{n} \left(\left\lfloor \frac{L - D_i}{T_i} \right\rfloor + 1 \right) C_i$$

Finally, a sufficient and necessary condition for EDF when deadlines are less than periods is given by:

$$\forall L \epsilon D: L \geq \sum_{i=1}^{n} \left(\left\lfloor \frac{L - D_i}{T_i} \right\rfloor + 1 \right) C_i \tag{8}$$

$$D = \{\, d_i^j \,|\, d_i^j = jT_i + D_i, d_i^j < LCM, 1 \leq i \leq n, j \geq 0 \,\}$$

i.e., we check intervals from 0 to each of the deadlines, until we either have checked intervals up to the LCM or the condition fails for some deadline.

Example: Is the following task set schedulable by EDF?

Task	C_i	D_i	T_i
τ_1	3	4	6
τ_2	4	7	8

Deadlines up to the LCM:

$$D = \{7,10,15,16,22,23\}$$

$$C^T(0,7) = \sum_{i=1}^{n} \left(\left\lfloor \frac{7 - D_i}{T_i} \right\rfloor + 1 \right) C_i = \left(\left\lfloor \frac{7-4}{6} \right\rfloor + 1 \right) 3 + \left(\left\lfloor \frac{7-7}{8} \right\rfloor + 1 \right) 4 = 7 \leq 7$$

$$C^T(0,10) = \left(\left\lfloor \frac{10-4}{6} \right\rfloor + 1 \right) 3 + \left(\left\lfloor \frac{10-7}{8} \right\rfloor + 1 \right) 4 = 10 \leq 10$$

$$C^T(0,15) = \left(\left\lfloor \frac{15-4}{6} \right\rfloor + 1 \right) 3 + \left(\left\lfloor \frac{15-7}{8} \right\rfloor + 1 \right) 4 = 14 \leq 15$$

$$C^T(0,16) = \left(\left\lfloor \frac{16-4}{6} \right\rfloor + 1 \right) 3 + \left(\left\lfloor \frac{16-7}{8} \right\rfloor + 1 \right) 4 = 17 > 16 \Rightarrow deadline\ miss!$$

Answer: The set is not schedulable; a deadline will be missed at time 16.

3.13 Comparison between Rate Monotonic and Earliest Deadline First

We will conclude this section about online scheduling by presenting a short comparison between EDF and RM, based on some selected criteria.

- RM and EDF have same implementation complexity – A small additional overhead is needed in EDF to update the absolute deadline at each instance release.
- RM is supported by commercial RTOSs – One big advantage of RM is that it can be easily implemented on top of fixed priority kernels.
- Runtime overhead is smaller in EDF – Due to the smaller number of context switches.
- EDF utilizes the processor better than RM – EDF achieves full processor utilization, 100%, whereas RM only guarantees 69%.
- EDF is simpler to analyze if $D = T$ – This is important for reducing admission control overhead in small embedded systems.
- Jitter reduction – EDF is fair in reducing jitter, whereas RM only reduces the jitter of the highest priority tasks.
- Aperiodic task handling – EDF is more efficient than RM for handling aperiodic tasks.

3.14 Offline scheduling

So far we have talked about online scheduling. We have learned about different techniques to schedule and analyze priority-driven real-time systems, both with statically and dynamically assigned task priorities. Now we will discuss clock-driven *offline* scheduling.

Offline scheduling is a method in which all scheduling decisions are pre-computed offline; before we start the system. The scheduler has complete knowledge of the task set and its constraints, such as deadlines, computation times, precedence constraints etc. The offline computed schedule is stored and dispatched later during runtime of the system.

Whenever the parameters of tasks with hard deadlines are known before the system starts to execute, a straightforward way to ensure that they meet their deadlines is to construct an offline schedule. This schedule specifies exactly when each task instance executes. According to the schedule, the amount of processor time allocated for each task instance is equal to its maximum execution time, and each instance completes before its deadline. As long as no instance ever overruns (i.e., some rare or erroneous condition causes it to execute longer than its worst-case execution time), all deadlines are surely met. Hence, we do not need any implicit analysis methods: if we manage to construct a schedule, then the system is schedulable. In other words, schedulability of the system is proven by construction. This makes offline scheduling suitable for usage in safety-critical systems (e.g., offline scheduling is used in the control system of Boeing 777).

Because the schedule is computed offline, we can afford to use complex, sophisticated algorithms to find the best schedule. We have time to do that, since the system is not started yet (it is not like in online scheduling where scheduling decisions are made at runtime, between clock ticks). Among all the feasible schedules, we may want to chose one that is good according to some criteria (e.g., the tasks are scheduled as close to their deadlines as possible, leaving place to include dynamic, aperiodic tasks). Hence, in offline scheduling we can solve more difficult scheduling problems than in online scheduling.

A straightforward way to implement an offline scheduler is to store the pre-computed schedule as a table; hence the name table-driven scheduling. Each entry (t_k, τ_i) in the table indicates the time t_k at which an instance of the task τi is released. During initialization, the operating system creates all the tasks that are to be executed. In other words, it allocates a sufficient amount of memory for the code and data for each task and brings the code executed by a task into memory.

The scheduler uses a timer, which is set to zero after the initialization. The timer generates interrupts with a certain periodicity (which depends on the timer granularity). This interrupts wakes up the scheduler, which is given the processor with the negligible amount of delay.

It is not practical to create a big schedule for each clock tick during the system evolution. Instead, we identify cycles that repeats them after a certain time. We use the same LCM approach as for online scheduling: the length of the cycle is equal to the least common multiple of all tasks.

Precedence constraints

We will see how we can construct offline schedules, but first we need to learn about a certain relationship between tasks – *precedence* relation.

In some real-time applications, computational activities cannot be executed in arbitrary order but have to respect some precedence relations defined at the design stage. If a task τ_i *precedes* a task τ_j then both tasks run with the same period, but τ_i must complete before τ_j starts to execute.

For example, assume a small control system in which we need to sample the environment with the rate of 1000 Hz and control and actuate at 100 Hz. In this case, we can specify a precedence relation between the control task and the actuator task so that the control task always sends the latest calculated value to the actuator task.

Precedence relations between tasks are usually described through directed graphs, called *precedence graphs*, PG, where tasks are represented by nodes and precedence relations by arrows, see Figure 14 for an example.

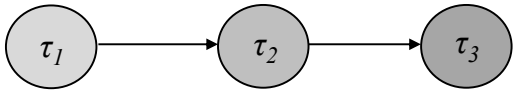
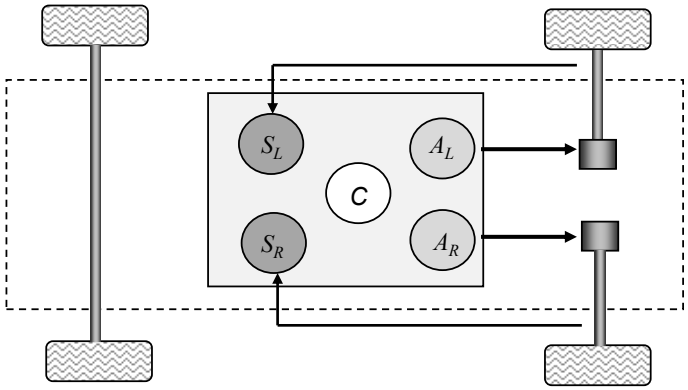


Figure 14: Example precedence graph.

In order to understand how precedence graphs can be derived from tasks' relations, let's consider the following example. Assume that we want to steer a simple toy car along a predefined path. Turning left/right is achieved by increasing/decreasing the rotation speed of respective front wheel, see Figure 15:



$\boldsymbol{S_L}, \boldsymbol{S_R}$ = read left/right rotation sensor

$\boldsymbol{C}$ = read the assigned path and calculate the rotation speed for the wheel motors

$\boldsymbol{A_L}, \boldsymbol{A_R}$ = send control values (actuate) wheel motors

Figure 15: Example steering of a robot car along a predefined path.

From the logical relations existing among the computations, tasks S_L and S_R, whose objective is to sample the rotation sensor from associated wheels, can be executed in parallel, since they do not depend on each other. Even two actuator tasks, A_L and A_R, which send control values to the wheel motors, can be executed in parallel, but each task cannot start before the computation of the new control values is done, in task C.

In order to get the latest computation values, task C must be executed with the same periodicity as tasks A_L and A_R, say for example 20. On the other hand, sensor task can run with higher frequency, having the shorter period, say 10. The precedence graphs (one for each period) look like presented in Figure 16:

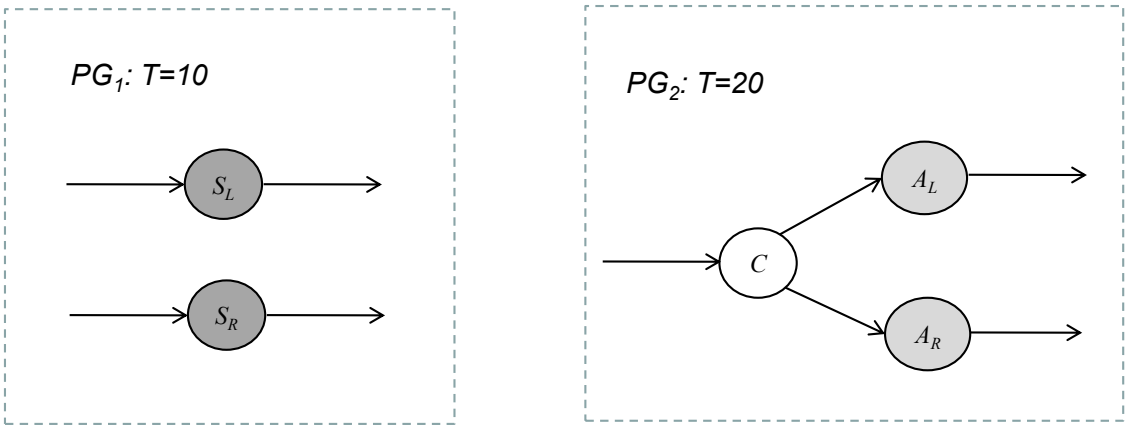


Figure 16: Example precedence graphs for a robot car.

Offline scheduling algorithm

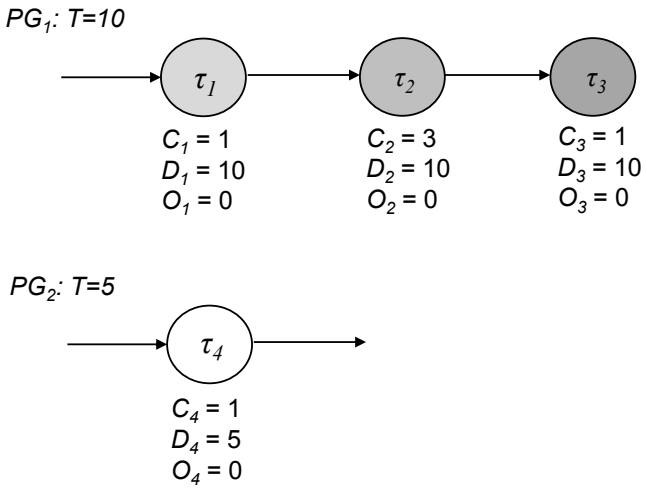
We will now describe a method to find an offline schedule from precedence graphs. The method consists of three steps:

1. *Joint graph* – We start by constructing a joint graph with period equal to the least common multiple, LCM, of all precedence graph's periods. Each precedence graph with a period T is inserted in the joint graph LCM/T times. For example, if the period of a precedence graph is 5, and the period of the joint graph is 15, then the precedence graph is inserted into the joint graph 15/5=3 times. Then we need to adjust release times and deadlines of the tasks in each of the inserted graphs.

2. *Search tree* – The next step is to generate a search tree from the joint graph. The search three contains all possible solutions (schedules) that can be constructed from the joint graph. If we recall the previous example with sampling tasks S_L and S_R, we can either run S_L first, then S_R, or the opposite, S_R first, then S_L. This means that two different (valid) schedules can be generated. If there are many such tasks that can run in arbitrary order relative each other, then the search tree gets many branches, each resulting in different schedules.

3. *Feasible schedule* – Traverse the search tree to find a solution. If the search tree is big, it can time quite a long to find a solution. Besides, not all possible schedules might be equally good for the application. Hence, here we can use some heuristics to minimize searching time, or to obtain a schedule according to some criterion.

The method is best explained through an example. Assume two precedence graphs, one having the period 10 and consisting of three tasks, and the other one with period 5 and only one task, as illustrated bellow (Parameter O is the offset, i.e., the activation time of the task counted from the period start):

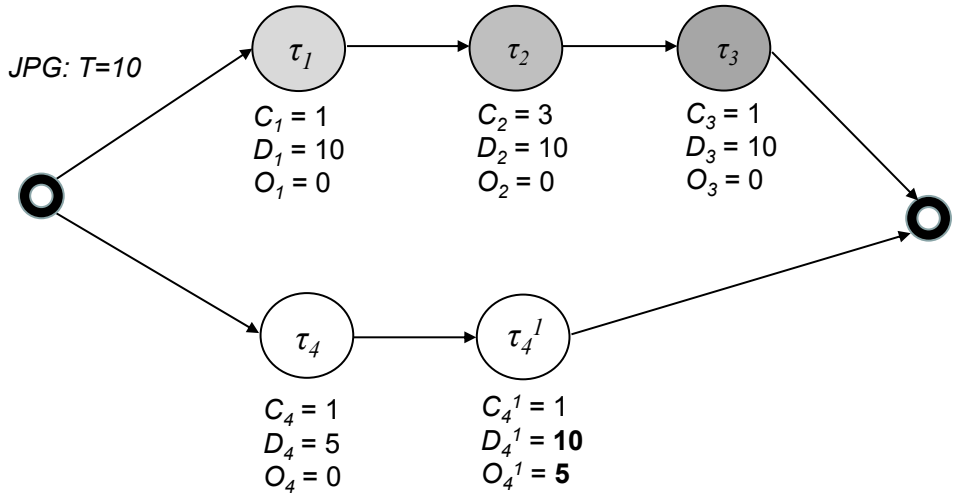


We will apply the three steps of the method to obtain a valid offline schedule.

Step 1: Construct a joint precedence graph, *JPG*. We start by calculating the period of the joint graph, which is equal to the least common multiple of the periods of all precedence graphs:

$$T_{JPG} = LCM(T_{PG1}, T_{PG2}) = LCM(10,5) = 10$$

PG_2 has a period of 5, which means it needs to be inserted into the joint graph twice. Hence, all tasks in PG_2 (i.e., task τ_4), will be repeated twice. Therefore we need to create a new instance of τ_4, and insert it in the graph:

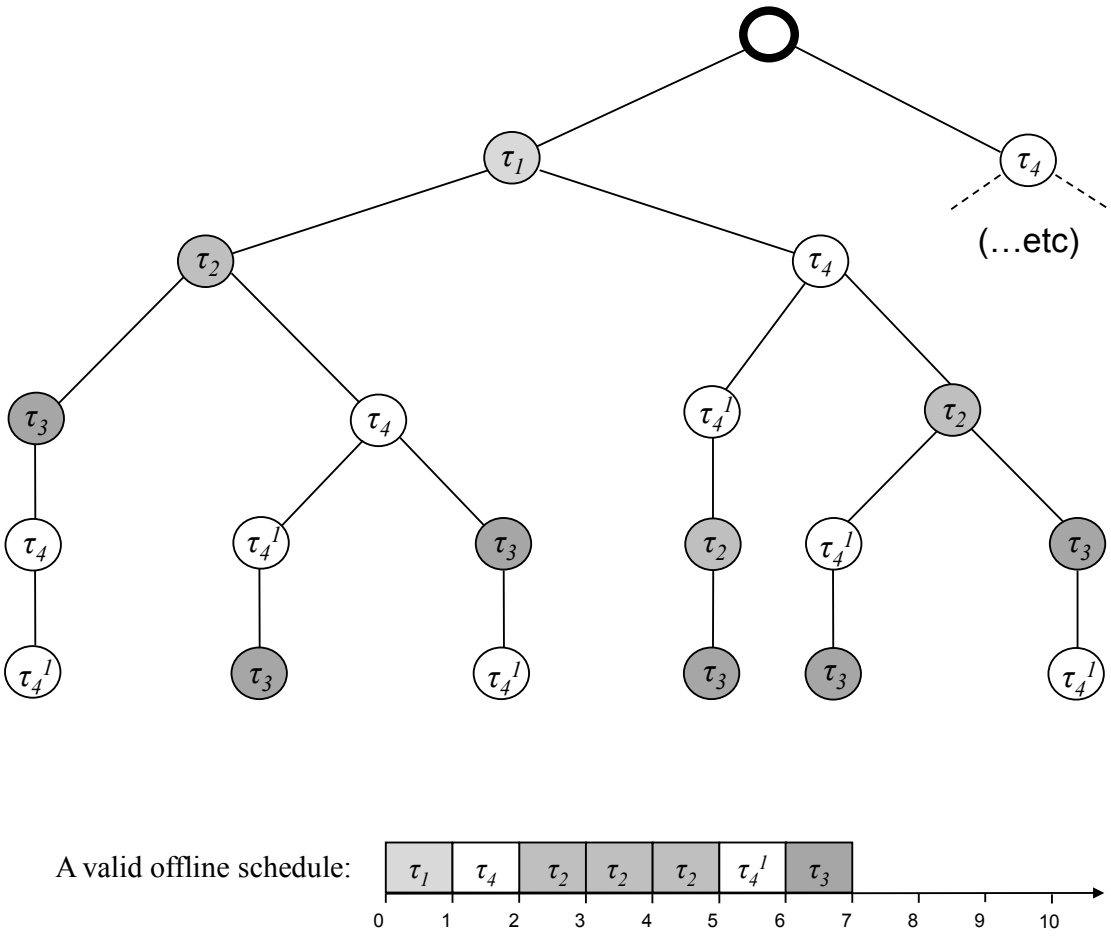


Notice changed release time and deadline for the new instance of τ_4. The reason is simple: if the first instance of τ_4 is released at time 0, then the next one cannot be released until the next start of the period for task τ_4, which is 5. Hence, the first instance of τ_4 will have release time 0 and deadline 5, while the second one will be released at time 5, and it will have deadline 10. If we need to create one additional instance of τ_4, it would get the release time 10 and deadline 15, and so on.

All tasks of PG_1, on the other hand, have the same period as the period of the joint graph, i.e., all tasks in PG_1 will be repeated once during the hyper period. Hence, we do not need to create any additional instances of tasks τ_1, τ_3 and τ_3.

Step 2: We generate search tree by traversing the joint graph. There are two possible paths from the start of the joint graph: we can either take the path $\tau_1 \rightarrow \tau_2$ or $\tau_1 \rightarrow \tau_4$. If we chose the first one, the next choices are either $\tau_1 \rightarrow \tau_2 \rightarrow \tau_4$, or $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$, and if we pick the second path, then the choices are $\tau_1 \rightarrow \tau_4 \rightarrow \tau_2$, or $\tau_1 \rightarrow \tau_4 \rightarrow \tau_4^1$, and so on, as showed in the figure below.

Step 3: We traverse the search tree to find a valid schedule. The search tree will contain all possible solutions to the scheduling problem. For example, we can find a feasible schedule if we follow the path $\tau_1 \rightarrow \tau_4 \rightarrow \tau_2 \rightarrow \tau_4^1 \rightarrow \tau_3$.



As we can see above, all task release times and deadline are met.

However, the schedule above is not the only solution, e.g., even the path $\tau_1 \rightarrow \tau_2 \rightarrow \tau_4 \rightarrow \rightarrow \tau_4^1 \rightarrow \tau_3$ gives a valid schedule. The search tree also contains all non-feasible (invalid) schedules as well. For example, if we take the path $\tau_4 \rightarrow \tau_4^1 \rightarrow \tau_1 \rightarrow \tau_2 \rightarrow \tau_3$, then the produced schedule will not be valid, since task τ_2 will miss its deadline.

Heuristic function

In the method above, the size of the search tree increases exponentially for each new task we add to the system, which means it can take very long time to find a solution. Moreover, if we allow preemption between tasks, then there is a higher probability that we will find a feasible solution, but it also implies larger search tree. Hence, we need to use some strategy, *heuristic*, to decrease the searching time for a valid schedule.

The idea with a heuristic function is to make decision which task (path) to chose, whenever we need to chose between two tasks. For example, chose the task with the shortest deadline, or, another heuristic could be to chose the task with the longest execution time.

A heuristic function should return a solution as soon as possible. It also needs to, as soon as possible, return the negative answer if no solution can be found.

If we use heuristics in the previous example, we will obtain a solution right away. For example, if we use "earliest deadline" heuristics, we will chose task τ_4 over task τ_1, since τ_4 has shorter deadline, 5. Then we will chose task τ_1 (notice that τ_4^1 does not compete with τ_1 because it has release time 5, while τ_1's release time is 0). Hence, if we continue this way, we would end up with a valid solution on the first attempt.

Mutual exclusion

Mutual exclusion between two tasks is pretty straightforward to implement in offline scheduling. If preemption is not allowed, then this problem is solved automatically by the scheduler: if tasks do not preempt each other, they cannot access same resources simultaneously. If we allow preemption, we can provide mutual exclusion in the scheduler by separating in time the tasks that share resources. This means that we must instruct the offline scheduler to check that tasks accessing same resources are scheduled in such way they cannot preempt each other. We can do that by automatically rejecting such branches in the search three which lead to interleaved execution of the tasks with shared resources.

3.15 Summary

In this chapter, we presented different techniques for online and offline real-time scheduling. In online scheduling, all scheduling decisions are taken at run-time, based on task priorities, while in the offline scheduling a schedule is created before run-time.

In online scheduling, the problem of scheduling a set of independent and pre-emptable periodic tasks has been solved both under fixed priority and dynamic priority assignment. The Rate Monotonic algorithm is optimal among all fixed priority assignments, while Earliest Deadline First algorithm is optimal among all dynamic priority assignments. When deadlines are equal to periods, the schedulability test for both algorithms can be performed with polynomial complexity, i.e., $O(n)$, by calculating processor utilization and comparing it to an upper bound . The schedulability test for RM, however, provides only sufficient condition for guaranteeing the feasibility, while it is both sufficient and necessary for EDF.

In the general case, in which deadlines can be less or equal to periods, the schedulability analysis becomes more complex. Under static priority assignments, the schedulability test is performed by using Response Time Analysis, which uses a recursive formula to calculate the worst-case finishing times of all tasks. Under dynamic priority assignments, the feasibility can be tested using the Processor Demand Approach. In both cases, the test provides a sufficient and necessary condition.

One big advantage of RM over EDF is that it is supported by many commercial RTOSs, which are all priority based. On the other hand, EDF is a more efficient scheduling algorithm with respect to effective processor utilization, schedulability analysis, jitter reduction, aperiodic task handling and run-time overhead. One of the challenges of today's real-time research is to develop EDF kernels to exploit all the advantages of dynamic scheduling without paying additional overhead.

In offline scheduling, a pre-runtime scheduler analyses the constraints between tasks, e.g., precedence constraints, and creates an offline schedule. There might be several solutions, so the time to produce a valid schedule can be reduced by providing an heuristic function when searching the valid schedule paths.

Offline scheduling is usually used in systems where there are high demands on timing and functional verification, testability and determinism, e.g., in safety-critical applications. Online scheduling is usually deployed when there are demands on flexibility, and many non-periodic activities, such as aperiodic and sporadic tasks.

3.16 Exercises

1. Discuss the difference between the following categories of scheduling algorithms:

   a) Offline vs online scheduling algorithms
   b) Static vs dynamic priority assignment algorithms
   c) Pre-emptive vs non-pre-emptive scheduling algorithms

2. Verify the schedulability by *Rate Monotonic* of the task set below and draw the schedule:

Task	C_i	$D_i=T_i$
τ_1	2	6
τ_2	2	8
τ_3	2	12

3. Assume a real-time system with n independent tasks, where the tasks have different, non-harmonic periods $T_1,T_2,\ldots T_n$, which are equal to the deadlines, and all tasks have the same execution time C, i.e., $C=C_1=C_2=....=C_n$. What is the maximum value of C such that the set is schedulable by using Rate Monotonic?

4. Analyze an online scheduled real-time system consisting of two tasks:

   - task L, which has the lowest priority and it is time-triggered, and
   - task H, high priority, event-triggered

   Task L regulates the rotation speed of a motor and it has the period of 50 ms and the execution time of 10, while task H reacts on the changes (events) from user (e.g., increase/decrease speed), and it has the execution time of 3 ms.

   a) How frequently can two events that H handles occur, such that the system is still schedulable?
   b) What is the maximum and the minimum time distance between completion (finishing) times of two consecutive instances of L?

5. Determine if the task set below is schedulable by Rate Monotonic by applying appropriate schedulability analysis:

Task	C_i	$D_i=T_i$
τ_1	4	17
τ_2	6	34
τ_3	12	68
τ_4	16	68

6. Assume a real-time system with seven periodic tasks that share resources:

Task	C_i	D_i	T_i	S_k	$cs(S_k)$
τ_1	3	20	1000	S_1	2
				S_2	2
τ_2	10	100	100	S_2	7
				S_3	5
				S_4	2
τ_3	20	50	50	S_2	2
τ_4	5	10	57	S_1	1
τ_5	1	33	33	-	-
τ_6	1	7	7	-	-
τ_7	2	5	30	S_1	1

Assume *Deadline Monotonic* (DM) priority assignment for the tasks.

Assume *Priority Ceiling Protocol* (PCP) for semaphore access.

Calculate worst-case response times of the tasks.

7. Assume the following set of periodic tasks that are to be scheduled by *Earliest Deadline First* (EDF) scheduling:

Task	C_i	$D_i=T_i$
τ_1	1	2
τ_2	1	3
τ_3	1	5

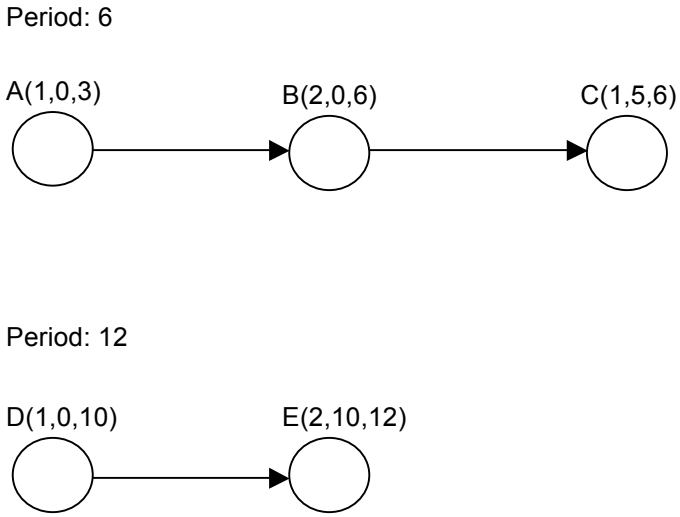
a) Determine if the task set above is schedulable EDF by applying appropriate schedulability analysis test.

b) If the set is schedulable, draw an execution trace up to the LCM of the tasks periods. Otherwise, if the set is not schedulable, draw a trace until the first deadline miss.

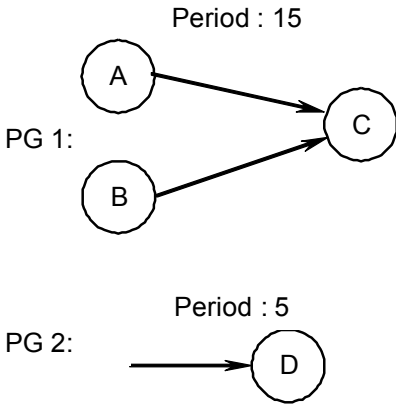
8. Is the following task set schedulable by EDF. Motivate your answer by performing appropriate schedulability analysis.

Task	C_i	D_i	T_i
τ_1	1	2	3
τ_2	1	2	4
τ_3	2	4	5

9. Use pre-emption and earliest deadline as heuristic function to offline schedule following tasks, where a task is specified as: name(execution time, offset, deadline)



10. Create a feasible offline schedule.



Task	Exe time	Release time	Deadline
A	4	0	7
B	3	0	12
C	5	0	15
D	1	3	4