17. Real-Time Scheduling

So far we have been considering the problem of scheduling jobs/tasks onto resources by considering fairness of resource assignment and "average" response time minimization. This approach is generally fine if we consider applications for which the concept of time does not play a role in their correctness. Take as an example the computer you deal with every day. When you open an application, the time it takes for the application to load and for you to be able to interact with it heavily fluctuates. Sometimes it can take a split-second, while other times it can take tens of seconds.

There are three main reasons for this behavior. First, the guy (or more likely the team) who coded that application mostly did not really care about how long it would take for the application to load. Second, your OS does not really put any constraint in how it should take for the application to load. Third, manufacturer(s) who made the hardware of your PC tried to make it as fast as possible. In general, the application developer, the OS and the hardware try to optimize the **average-case** performance of your system. However, they think very little about the **worst-case** performance of your system. And you know how much those two can differ!

Now, reasoning about the average case performance is mostly fine for general-purpose applications. This is because applications and computing systems in general do not "feel" time. In fact, time is a property of the physical world. You have probably noticed that when you code an application, you never have to worry about timing in strict sense. Often you worry about "complexity" of an algorithm. But the same algorithm can execute in milliseconds on the latest-generation Intel processor, while it may take several days on a micro-processor from the early '90s. Often in your applications you may have to worry about a timeout. Timeouts can occur when our application interacts with something "physical": a network, a disk, a user, and so on.

17.1 Cyber-Physical Systems

So there you have it. As long as applications do not have to interact with the physical world, they can take their sweet time to execute. But what if an application is **in charge** of what happens in the physical world? If you think that this is a rare occasion, be prepared to be surprised. In fact, every single day of your life, you interact with systems where there is an application in charge of a physical process. Here are some examples:

1. The thermostat in your home. Somewhere in that little box attached to your wall there is a micro-controller. That controller monitors the temperature in your home and turns on/off your heating system accordingly.
2. The washing machine that you use for your laundry is yet another example: a processor controls the temperature of the water, the rotation speed of the cylinder, as well as the amount of time your clothes will spin around.
3. The Charlie Card reader. The reader is connected to a processor that can decode the info in your card and that, in few milliseconds, tells you if you have enough credit to board the bus/train.
4. Do you drive to campus? Then you are literally surrounded by this kind of systems. You have systems that read the steering position and apply steering to the wheels accordingly. In the unlucky case of an accident, your best friend is the Airbag Control Unit (ACU), that will quickly deploy the airbag.

Because these are systems that perform computation but also interact tightly with the physical world, the term **cyber-physical systems** (CPS) is used. Cyber-physical systems is a very generic term that refers to a broad class of systems who: (1) sense the physical world via a number of **sensors**; (2) perform sensor fusion and enact a **control** strategy; (3) use **actuators** to interact with and change the physical world.

The term CPS is used to describe a class of very different systems, as it is clear from the list above. These systems mostly differ on how crucial is the physical process that they are in charge of. Clearly, if the Charlie Card reader does not respond after tens of seconds, you will get mildly frustrated, but mostly likely will score a free ride. So not that much of a big deal. Conversely, if you have an accident and your airbag does not deploy, the danger is very real.

In other words, we often classify CPS based on how catastrophic are the consequences in case of system's misbehavior. Typically, if the misbehavior of a system can jeopardize the safety of one or more people, then the system is considered **safety-critical**. Instances of safety-critical systems are: airplanes and air traffic control systems; electric power grid management systems; nuclear plant monitoring and control systems; electronic stability control (ESC) systems; flight controllers on commercial drones; the fuel injection system in an internal combustion engine. The rule of thumb is: if the misbehavior of a system can put someone's life at risk, then the system is safety-critical. Clearly, there is a controversy here on how we should classify advanced warfare systems. Is their purpose to induce or prevent loss of human lives?

17.2 Real-Time Applications

The term *real-time* is used to describe a programming model where the notion of physical (real) time, directly impacts the correctness of the system. Typically, cyber-physical systems are real-time systems. In order to distinguish between systems that are safety-critical and systems that are not, we use the term **hard real-time**, as opposed to **soft real-time**.

Why so much terminology? Because we need to figure out what model applies to which type of system. All we are saying is that if we intend to reason about safety-critical systems, we better use results for hard real-time systems. If your system is not so crucial after all, we can have a more soft real-time approach.

Computation processes in real-time systems are subject to time **deadlines**. Clearly, the deadline represents a constraint on how we schedule processes in a real-time system. Long story short, it is *okay* yet discouraged for a process to miss a deadline in a soft real-time setting; it is absolutely NOT okay to miss a deadline for a hard real-time task.

Where do deadlines come from? And how can we decide what is the deadline for a given task/job? For this, let us use an example. Let us consider again the airbag control unit (ACU). When a crash occurs, the driver is projected forward – within the abruptly decelerated car body – due to conservation of momentum. The ACU monitors a wide range of sensors (e.g. accelerometers, impact sensors, pressure sensors) to determine the occurrence of a crash. From the moment of the impact, the ACU has between 20 and 30 milliseconds to initiate an airbag deployment, that typically completes within 80 milliseconds. The goal is to gradually decelerate the forward-projected driver. If the airbag deployment is initiated **too slowly**, the forward-projected driver is impacted by a deploying airbag instead. The result would not be pretty.

If follows that the deadline arises because of certain dynamics of the underlying physical system being controlled. In the case of an ACU, the deadline depends on the *time* it takes for the driver's body to impact the steering wheel. So one may think that the solution for problems in the real-time domain is: let's complete our tasks as fast as possible. That does not work. For instance, in case of a ACU, if airbag deployment is initiated **too early**, the airbag will have started to deflate by the time the driver's body reaches the steering wheel, decreasing the effectiveness of the system.

17.2.1 Predictability

The last consideration brings us to a concept that is crucial in real-time systems: the notion of predictability. A system is said to be **predictable** if the difference in time between (A) its behavior in the best possible case, and (B) its worst-case behavior is small.

As you already know from the discussion at the beginning of this note, an application can take shorter or longer to execute. In fact, one can see the execution time of an application as a probability distribution, as depicted in Figure 17.1. For a given real-time task in a system, we can derive three main parameters that describe its timing behavior:

- **BCET:** best-case execution time. The time it takes for the task to execute if all the resources it

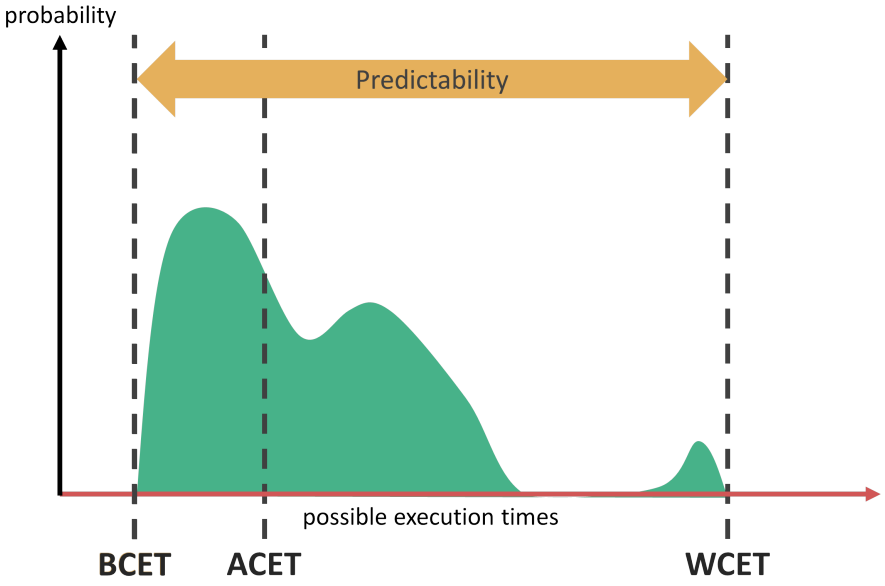

Figure 17.1: Distribution of execution time for a generic application. The difference between WCET and BCET indicates how predictable is the considered application.

requires take the least possible amount of time to satisfy the task's needs. E.g. there are no pipeline stalls, no cache misses, disk head for I/O operations is always in the right position, and so on.

- **ACET:** average-case execution time. The time it takes for the task to execute on average.
- **WCET:** worst-case execution time. The time it takes for the task to execute if everything goes horribly wrong. It has been found that WCETs occur more likely on Mondays[1].

Hence, a task is considered more predictable if the difference (WCET - BCET) is small. A system composed of tasks that are predictable, is itself considered predictable. A predictable system is easier to analyze, to reason about, and to certify for in-the-field operation. This also leads to an interesting conclusion. If there is not much we can do about the WCET of an application, it is okay (and encouraged) to trade in some performance to bring the BCET closer to the WCET. It follows that in real-time systems, performance is a secondary objective to "not missing deadlines".

It also follows that, when analyzing a real-time system, we will always reason in terms of WCETs. If we can prove that we are able to meet all the deadlines when all the tasks in the system exhibit their WCET, then we can be sure that our system will work well no matter what. Basically, real-time systems are analyzed according to Murphy's Law: *"anything that can go wrong will go wrong"*.

[1] Just kidding.

17.2.2 Deadlines, Tardiness, and Utility

As we mentioned earlier, the ideal behavior of a real-time system is that in which no deadlines are missed. Up until the deadline is missed, the utility of completing a job on time is positive and typically constant. What happens if a deadline is missed?

First of all, we define as **tardiness** the difference between the completion time of a real-time job and its deadline. Then we wonder: does it make sense to complete a job after the deadline has been missed? Or: how useful is to complete a job after a deadline has been missed? The answer to these questions is what distinguishes hard real-time from soft real-time systems.

In soft-real time tasks, the utility of completing a job after the deadline gradually decreases. For instance, decoding a video frame slightly after the ideal time-frame is *okay*. And the more tardiness is accumulated, the more the resulting video stream will lag.

Conversely, in hard real-time systems, the utility function drops immediately to zero after the deadline. A value of zero indicates that a catastrophic failure has occurred and that there is nothing (zero) the system can do. The utility could even become negative. The latter case refers to systems where completing a job after the deadline is even more dangerous. The latter is the case of the ACU: if after the impact the airbag did not deploy on time, and the driver has already hit the steering wheel, deploying the airbag is only (very!) harmful. Same for a bomb deploying mechanism: if we have missed the intended objective because the bomb did not deploy on time, it is better not to deploy the bomb at all, instead of hitting a random location.

There is one big problem with this classification: the utility function is subjective.

17.3 Real-Time Scheduling

Now that we know what are the characteristics and requirements of a real-time workload, we can re-think the role of the scheduler. In fact, the scheduler has to first meet the deadline requirements of real-time tasks. In doing so, it can also attempt to optimize resource utilization, overall predictability, and so on.

In other words, real-time scheduling is an instance of a constraint optimization problem. This objective formulation is very different from the problem of scheduling that we have considered so far, i.e. in case of general-purpose/non-real-time systems. In non-real-time systems, in fact, a good measure of performance was the ability of the system to minimize the response time of requests. Conversely, in real-time systems, it does not matter how long it will take for a job to complete, as long as its completion happens before its deadline[2].

A good way to measure the "performance" of a real-time scheduler is to measure the maximum amount of work that can be handled without missing any deadline. We know that as the utilization of a resource increases, the response time of submitted jobs also increases. Hence, there is a cap after which adding a new job in the system will lead to some job missing its deadline. This cap is

[2]Once again, performance intended in the traditional sense matters less than predictability!

Task	Period T	WCET C
τ_1	4	1
τ_2	10	4
τ_3	12	3

Table 17.1: Summary of job parameters used to illustrate the considered scheduling algorithms.

called the **utilization bound**. A real-time scheduler with higher utilization bound is to be preferred (in principle) because on the same processor we can schedule more workload without missing any deadline.

In order to reason about the ability of a scheduler to meet the deadline requirements of a given task-set, it is important to perform *schedulability analysis*. A schedulability analysis takes in input the parameters of the considered real-time task-set, considers the scheduling strategy in use, and outputs whether or not all the deadlines can be met.

17.3.1 Real-Time Task Model

Since real-time tasks are typically linked to sense-control-actuate problems, their behavior is inherently periodic. Servo-motors in commercial drones actuate at 50 Hz (i.e. every 20 milliseconds); the control loop in a helicopter is 180 Hz; and so on.

As such, each real-time task τ_i is given the following parameters:

- T_i: the period at which a new job $j_{i,k}$ of task τ_i is released. As such, new jobs of τ_i will be released every T units of time, and the release time of $j_{i,k}$ will be $k \cdot T$.
- C_i: the WCET of a job of task τ_i;
- D_i: the deadline by which a job of τ_i needs to complete, relative to its release time. If a job $j_{i,k}$ was released at t, its absolute deadline will be at $t + D_i$.

From now on, we will assume that a given job needs to complete before another job of the same task is released. I.e. we will assume that $D_i = T_i$. This task model is referred as *implicit-deadlines periodic task model*.

Figure 17.2 depicts an example of an implicit-deadline periodic task-set composed of two tasks. The parameters for the task-set used in Figure 17.2 is given in Table 17.1 and correspond to τ_2 and τ_3 in the table.

17.4 Static Priority Scheduling

As we mentioned when we discussed scheduling in the context of general-purpose systems, scheduling boils down to **assigning priority** to ready jobs. The ready job with higher priority is selected next to execute on the processor.

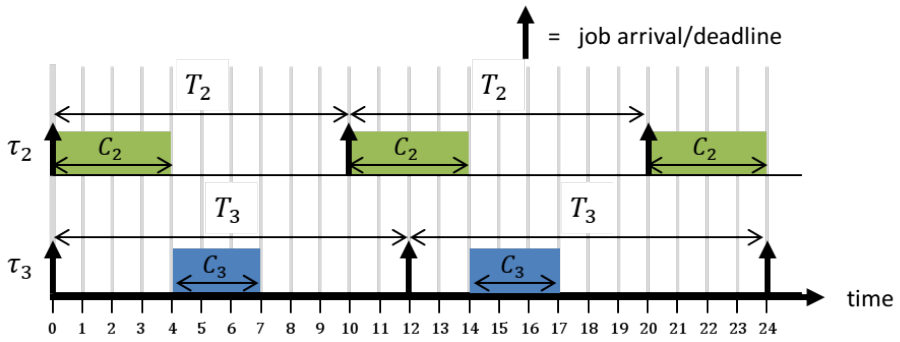

Figure 17.2: Visualization of task parameters for τ_2 and τ_3 reported in Table 17.1.

One possible option to perform priority-to-job assignment is to **statically** assign a priority to each task τ_i in the system. Whenever a job $j_{i,k}$ of τ_i is ready, it will inherit the priority of the "parent" task.

In this case, the scheduling problem can be formulated as follows: according to what strategy we can assign a priority to a task-set of m real-time tasks, such that no job released by the tasks $\tau_1, \ldots, \tau_m$ misses its deadline?

17.4.1 Rate Monotonic Scheduling

A popular strategy to assign static priorities to real-time tasks to optimize schedulability is **Rate Monotonic** (RM). Under RM, a task is given a priority that is inversely proportional to the length of the period. The lower the period (higher rate) of a task T_i, the higher its priority. If the periods of the tasks are all different, there will never be any tie. Otherwise, an arbitrary tie-braking strategy will do.

- **Invocation:** the scheduler is invoked at the completion of a job, or when a new job is released (preemptive).
- **Policy:** the ready job with the shortest period (highest priority) executes next on the processor, eventually preempting an executing job.

With the task parameters provided in Table 17.1, Figure 17.3 depicts an instance of RM scheduling.

As can be seen, in the current setup, all the tasks meet their deadlines. What happens if we add one more task to the set? Well it depends on its parameters. Suppose that we add a new task τ_4 with parameters $T_4 = 20$ and $C_4 = 2$. Figure 17.4 depicts the resulting schedule.

In this case we say that the task-set without τ_4 is schedulable under RM, while the task-set that includes τ_4 is not schedulable under RM.

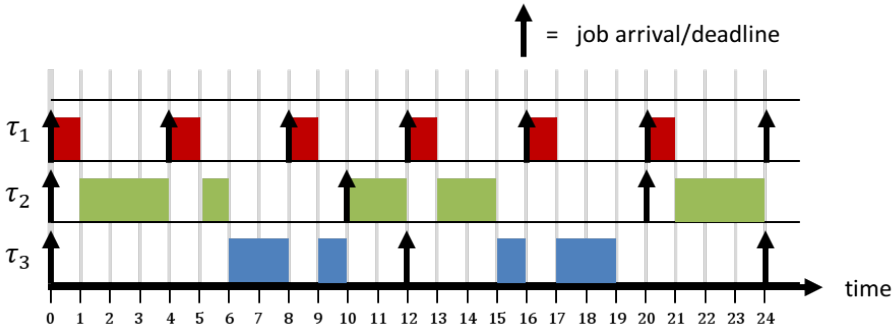

Figure 17.3: RM scheduling for task-set with parameters reported in Table 17.1.

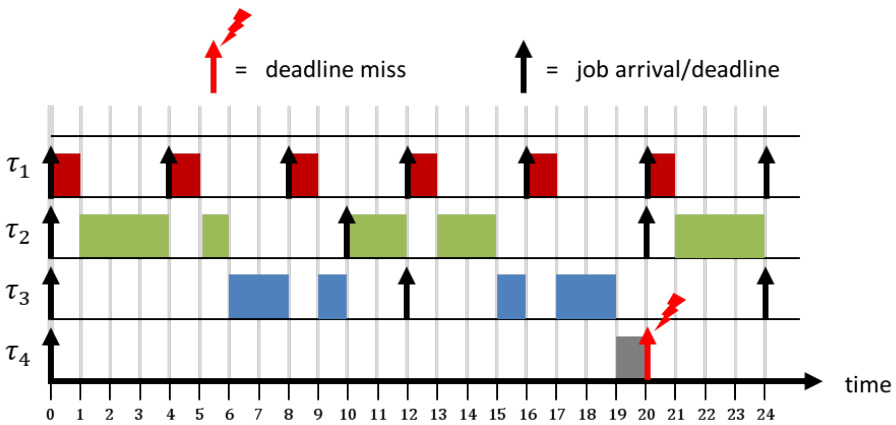

Figure 17.4: RM scheduling for task-set with parameters reported in Table 17.1 with the addition of τ_4 with parameters $T_4 = 20$ and $C_4 = 2$. Task τ_4 misses its deadline at $t = 20$.

RM Schedulability Test

Of course it is easy to tell if a task-set is schedulable after we have extracted the produced schedule like in Figure 17.3 and 17.4. The question is, however: can we tell whether or not a task-set is schedulable by just looking at the parameters? Is there a closed formula to compute schedulability. If such a formula exists for a given scheduler, that formula is called a **schedulability test**.

In case of RM, the following theorem comes in help and constitutes a schedulability test.

Theorem 17.4.1 A given task-set comprised of m tasks $\tau_1, \ldots, \tau_m$ is schedulable under RM if Equation 17.1 holds.

$$\sum_{i=1}^{m} \frac{C_i}{T_i} \leq m \cdot (2^{\frac{1}{m}} - 1) \tag{17.1}$$

The quantity expressed in the left-hand side of Equation 17.1 is called **utilization** of the task-set. How does this apply to the task-sets considered so far? Let us calculate their utilization.

For the task-set without τ_4 ($m = 3$) we have that utilization = $\frac{1}{4} + \frac{4}{10} + \frac{3}{12} = 0.9$. The right-hand side of Equation 17.1 becomes $3 \cdot (2^{1/3} - 1) = 0.78$. So Equation 17.1 **does not** hold. Re-read the theorem one more time: the expressed condition is sufficient, not necessary! This means that if you provide a task-set whose parameters satisfy Equation 17.1, then the task-set is certainly schedulable under RM. If the condition does not hold, **the test simply yields no conclusion**. In the latter case, the only solution to check the schedulability is to compute the worst-case completion time of a job in each task, for instance by drawing the scheduling diagram.

Thanks to Figure 17.3, we know that this particular task-set is schedulable despite its high utilization (90%!). Conversely, when we add task τ_4 in the taskset, the utilization reaches 100% and the task-set is not schedulable anymore under RM (see Figure 17.4).

We may now wonder, what is that value of utilization under which, no matter how many tasks we have, RM will always be able to produce a schedule without any deadline miss? If we can answer this question, we have found a **utilization bound** for RM. Recall that for real-time scheduling algorithms the utilization bound is a good way to express the performance of the scheduler.

Since we already have a schedulability test for RM, we start from there. In particular, we calculate the right-hand side of Equation 17.1 for an arbitrarily large m. When we do so, the value of $m \cdot (2^{\frac{1}{m}} - 1) \rightarrow \ln(2) = 0.69$. If follows that any task-set with utilization lower than or equal to about 69% is schedulable under RM. The trend of the expression $m \cdot (2^{\frac{1}{m}} - 1)$ for increasing values of m is depicted in Figure 17.5.

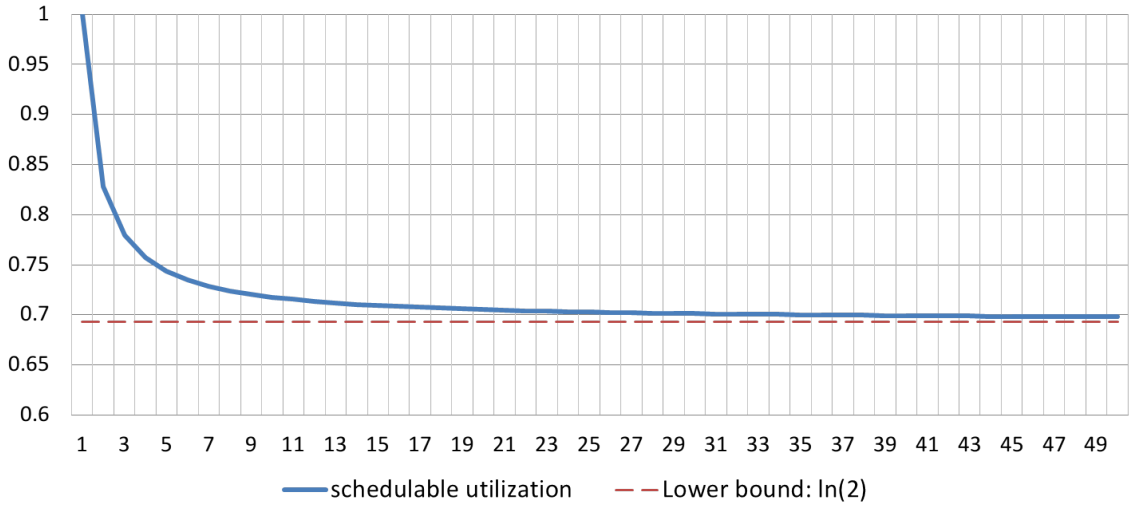

Figure 17.5: Trend of the value of $m \cdot (2^{\frac{1}{m}} - 1)$ for m in the interval $[1, 50]$. The value approaches $\ln(2)$ (dashed line) for large values of m.

17.5 Dynamic Priority Scheduling

Another approach for scheduling real-time tasks consists in re-computing and re-assigning the priorities of the ready tasks as they arrive, according to the current status of the system.

Intuitively, this allows for more "informed" decisions, and hence better schedulability. If a scheduler re-computes task priorities at the occurrence of scheduling events, we are in the presence of a **dynamic-priority** scheduling algorithm. In this case, there is no such task-level priority, because the priority of two different jobs of the same task can change. Some algorithms even vary the priority of the same job across time.

Let us now take a look at a popular scheduling strategy for real-time tasks on a single processors.

17.5.1 Earliest Deadline First

Earliest Deadline First (EDF) is a dynamic priority scheduler which assigns a priority that is directly proportional to how close is the deadline for a ready job. Since priorities change only when new jobs are released the scheduler only needs to be invoked at the released or completion (as always) of jobs.

- **Invocation:** the scheduler is invoked at the completion of a job, or when a new job is released (preemptive).
- **Policy:** the ready job with the closest deadline (highest priority) executes next on the processor, eventually preempting an executing job.

Contrarily to RM, even if all the tasks have different periods, it can still happen for two different ready jobs to have the same absolute deadline, and hence the same priority according to EDF. As such, it becomes important to break these ties. Once again, we can use the task ID to break these ties.

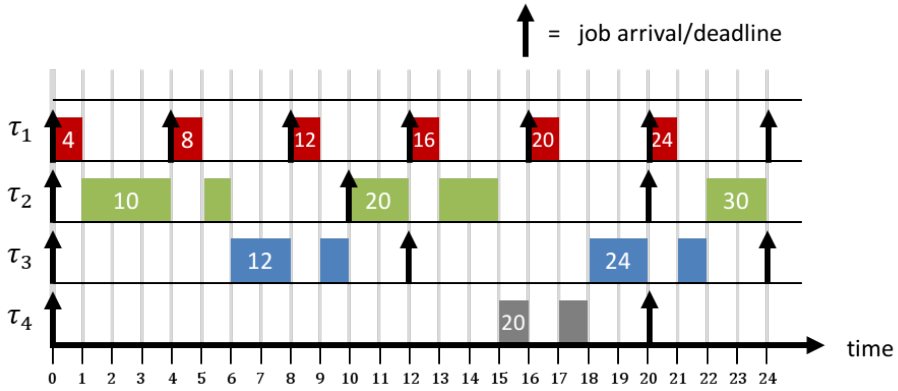

Figure 17.6: EDF scheduling for task-set with parameters reported in Table 17.1 with the addition of τ_4 with parameters $T_4 = 20$ and $C_4 = 2$. In case of ties, priority is given to the job with **lower** task ID. No deadline misses.

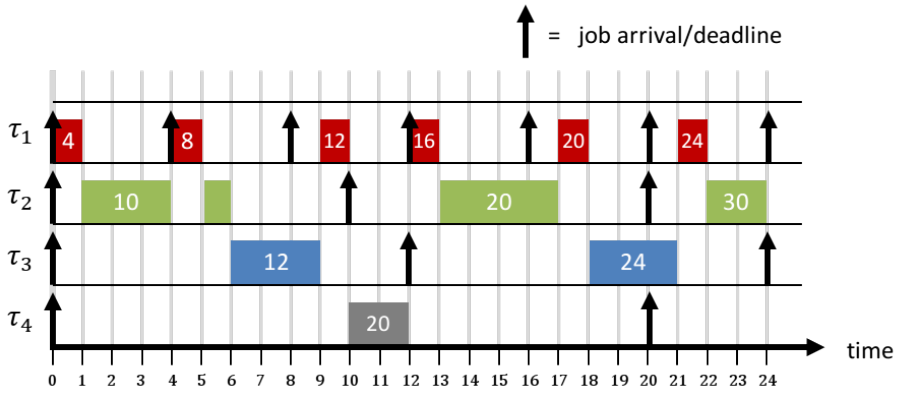

Figure 17.7: EDF scheduling for task-set with parameters reported in Table 17.1 with the addition of τ_4 with parameters $T_4 = 20$ and $C_4 = 2$. In case of ties, priority is given to the job with **higher** task ID. No deadline misses.

With the parameters provided in Table 17.1 and the addition of task τ_4, the resulting schedule produced by EDF is depicted in Figure 17.6. This is the schedule that corresponds to a tie-breaking strategy that gives higher priority to the job with lower task ID. As a reference consider the case when ties are broken bu giving higher priority to the job with higher task ID in Figure 17.7. For convenience, in both figures the absolute value for the deadline is reported on each released job. EDF will pick the ready job with lower absolute deadline (and break the ties as described above).

Once thing is pretty obvious: there are no deadline misses anymore with EDF scheduling, despite we have used the same task-set that resulted in a deadline miss under RM (see Figure 17.4). We can also note one more thing: if we compute the utilization of the task-set, we have: utilization = $\frac{1}{4} + \frac{4}{10} + \frac{3}{12} + \frac{2}{20} = 1$. In other words, EDF was able to schedule a task-set with 100% utilization. Is that a coincidence? The following theorem helps us out.

Theorem 17.5.1 A given task-set comprised of m tasks $\tau_1, \ldots, \tau_m$ is schedulable under EDF **if and only if** Equation 17.2 holds.

$$\sum_{i=1}^{m} \frac{C_i}{T_i} \leq 1 \tag{17.2}$$

This theorem clearly represents a schedulability test for EDF in a single-processor setting. There is a subtle difference compared to the schedulability test that we have seen for RM. In this case, in fact, the test expresses not only a sufficient condition, but a necessary one as well. If we are given a task-set and the test succeeds, we know that the task-set can be scheduled with EDF. If the test fails, however, we also know that the task-set will definitely be non-schedulable. The last conclusion is only valid for the EDF test, not for the RM one.

We also have one more result. Clearly, there is no way to schedule tasks without missing deadlines if their utilization goes beyond 100%, no matter which algorithm we use. Since EDF is always able to schedule task-sets with utilization up to 100%, then EDF is the best we can do. As such, EDF is said to be **optimal** for preemptive real-time tasks on single-processor.

17.6 Practical Scheduling

Now that we know two of the main algorithms to perform schedulability of real-time tasks on single-processors, we can try to draw some conclusions on what is best suited for use in real systems.

One could rush and say that because EDF is able to use the processor more efficiently without missing any deadline, then EDF is to be preferred. History teaches us that it is not the case. There are actually few important reasons why RM represents the de-facto standard in truly hard real-time / safety-critical systems.

The first reason is **complexity**. With RM all we have to do is assigning an integer value offline for each task in the system, that can simply be priority of task $i = -T_i$. At runtime, when a new job is released or an old one is completed, the OS picks the first ready task with higher priority. Done. Conversely, with EDF the scheduler needs to compute the absolute deadline of the arriving task and decide if the newly calculated priority is higher or lower than the currently running task. Per-se, the operation is not that complicated, but the additional complexity in the OS scheduler is not really all the worthy, as we will see below.

One may think that utilizing 30% more of the available processor capacity is indeed a big deal, and would justify the additional complexity of implementing EDF. There are two issues with that. The first is that not all the workload in a real system is really real-time. Systems perform a lot of book-keeping, data buffering, self-diagnostic and so on. These operations use processor time, but do not really have a deadline. Hence, the solution is to use RM for all the real-time tasks and use the leftover processor time for all the non-critical applications. Secondly, it is possible for RM to achieve 100% schedulability if the parameters of the tasks satisfy certain criteria. So if you really want 100% real-time workload utilization, it is enough to have real-time tasks with nice enough parameters.

Last but least, EDF has a big problem when it comes to jobs that could potentially overrun their WCET (due to an unexpected malfunction). In a 100% loaded system, this would lead to a deadline miss under both RM and EDF. This per-se is bad enough. But with EDF it gets worse. Recall that the priority of a job under EDF is calculated as the distance between current time and job's absolute deadline. Then what is the priority of a job that has already missed its deadline? In this case, the behavior of EDF is undefined, leading to lack of system predictability, and a ripple effect that can affect potentially all the other tasks in the system (despite only one misbehaved). This problem is not present in RM, where a misbehaving task can only affect lower priority tasks.