UNIVERSITÀ DEGLI STUDI DI TRENTO

Department of Information Engineering and Computer Science

Master Degree in
Computer Science

Real-Time Operating System and Middleware

Professor
Luigi Palopoli

Assistant
Tadeus Prastowo

Academic Year 2020/2021

Contents

1 Introduction

Real-Time Applications are applications that have to comply with temporal constraints, so for example restarts are not even thinkable.
Real-Time Operating Systems are operating systems that provide support for Real-Time Applications.
An operating system can be seen from different perspectives:

- Set of critical computer programs: if they don't work correctly the system does not work;
- Interface between applications and hardware;
- Control the execution of application programs;
- Manage the hardware and software resources.

So basically an OS can be seen as a service provider for user programs since it exports a programming interface, or as a resource manager since it implements schedulers.
Resource, i.e. process, memory, I/O management etc., must be managed so that real-time applications are served properly. For such reason, the OS must implement schedulers, so that all the real-time processes can run.
The services the OS provides are:

- Process synchronization and inter-process communication;
- Process and thread scheduling;
- I/O;
- Virtual memory.

Instead, as a resource manager, the OS provides:

- Interrupt handling;
- Device management.

Since *real-time* applications will be the main focus, this implies that the time when a result is produced matters: since a correct result produced too late is equivalent to a wrong one, so this type of applications is characterised by *temporal constraints* that have to be respected. Notice that simply having a "fast" system does not mean that the constraints will be fulfilled, indeed, formally proving that the temporal constraints are always respected is the most important aspect. In the general case, we are interested in the throughput, that is how many operations are completed at a time. In the worst case scenario instead we are interested in understanding how many times such scenario happens.
Let's recall some definitions that will be used later on:

- **Algorithm**: it's a logical procedure used to solve a problem;

- **Program**: it's a formal description of an algorithm using a programming language to describe it;
- **Process**: it's the instance of a program, i.e., the program when in execution;

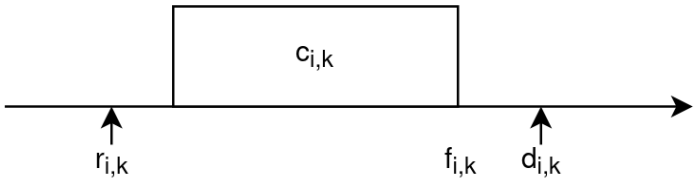
2 Real-Time Scheduling

2.1 Tasks

A real time task τ_i is a sequence of activities, called **jobs** or **instances**, $J_{i,k}$ where i is the i-th task and k is the k-th activation of the i-th task, that is the k-th job. Each job $J_{i,k}$ is defined by 4 values:

- $r_{i,k}$: the activation time, that is the moment in which the job is activated;
- $c_{i,k}$: the computation time, that is the time that it takes to execute;
- $f_{i,k}$: the finishing time, that is the time in which the job ends;
- $d_{i,k}$: it's an absolute deadline within which the job must finish, so we want $f_{i,k} \leq d_{i,k}$.

We'll write $J_{i,k} = (r_{i,k}, c_{i,k}, f_{i,k}, d_{i,k})$.



A job is basically an abstraction used to associate deadlines, which are temporal constraints, to activities:

- $r_{i,k}$: the time when job $J_{i,k}$ is activated. There can be different types of activation, for instance external events as sensors values, or time activation. Notice that the start of the task does not always coincide with the activation time since it's not given that the CPU will be free at that instant, so the start of the job could be delayed.
- $c_{i,k}$: *computation time* needed by job $J_{i,k}$ to complete. Even though jobs do the same things over and over, we give them different computation times because we could receive different inputs. Also cache effects should be kept into consideration: if we find the value inside the cache then it's faster than retrieving it from memory.
- $d_{i,k}$: absolute time instant by which job $J_{i,k}$ must complete. A job is said to respect the deadline if $f_{i,k} < d_{i,k}$.
- $\rho_{i,k}$: the *response time*, which is the time that passes between the activation time and the completion of the task:

$$\rho_{i,k} = f_{i,k} - r_{i,k}$$

2.1.1 Periodic Tasks

A periodic task $\tau_i = (C_i, D_i, T_i)$ is a stream of jobs, each one setting off at a predefined interval:

$$r_{i,k+1} = r_{i,k} + T_i$$

where T_i is the *task period*. Moreover each deadline is spaced out by a fixed amount of time

$$d_{i,k} = r_{i,k} + D_i$$

where D_i is the *relative deadline*, i.e., the time left before the absolute deadline $d_{i,k}$.

$$C_i = \max_k \{c_{i,k}\}$$

C_i is the *Worst-Case Execution Time* (WCET). Lastly, the *Worst-Case Response Time* WCRT is defined as:

$$R_i = \max_k \{\rho_{i,k}\} = \max_k \{f_{i,k} - r_{i,k}\}$$

For the task to be correctly scheduled, it must be:

$$R_i \leq D_i$$

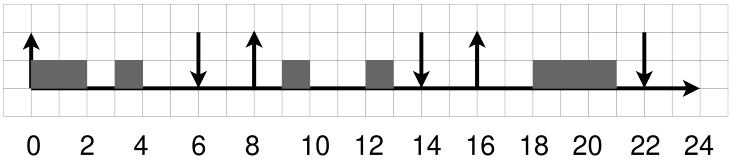
Instead of referring to all the different job, we compute the worst-case response time and say that until this is smaller than the relative deadline D_i, then we are fine.
So we want the periodic task to be activated periodically each T_i, to execute a computation and then to suspend waiting for the next period. A skeleton in pseudo-code could be the following:

```
1  void *PeriodicTask(void *arg){
2    //Initialization
3    //Start periodic timer, period=T_i
4    while (cond) {
5      //Read sensors
6      //Update outputs
7      //Update state variables
8      //Wait for next activation
9    }
10 }
```

Notice that the outputs are update before the internal variables because we want to give the answer as soon as possible.

2.1.2 Graphical Representation

Tasks are graphically represented by using a **scheduling diagram**.
For example, the following picture shows a schedule for a periodic task $\tau_1 = (3, 6, 8)$:



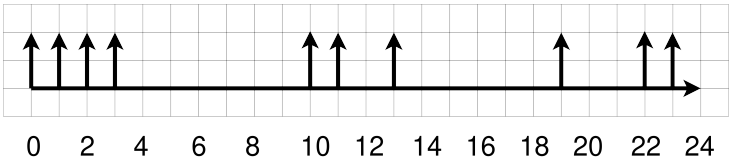
Upward arrows indicate activation of the task, while downward arrows indicate termination. The gray boxes indicates the period of time for which the task is running.
Notice that while jobs $J_{1,1}, J_{1,3}$ executes for 3 units of time which is actually the worst-case scenario, job $J_{1,2}$ executes for only 2 units of time, moreover it's possible to see that jobs $J_{1,1}, J_{1,2}$ are interrupted, probably because the CPU was "given" to some other tasks.

2.1.3 Aperiodic Tasks

Aperiodic tasks are not characterised by periodic arrivals, hence there is no minimum inter-arrival time between activations.
We have seen a basic programming structure for periodic tasks, but such structure does not exist with aperiodic tasks.
This type of tasks can model:

- Tasks responding to events that occur rarely, for example a mode change;
- Tasks responding to events with irregular structure, for example bursts of packets from the network.

So a possible representation of aperiodic tasks is the following:



It's possible to see that in some cases we have bursts of arrivals, while in some other we don't have arrivals at all. If the bursts are really frequent in the number of activations, no temporal constraint can be safely put in place since the execution time will probably be longer.

2.1.4 Sporadic Tasks

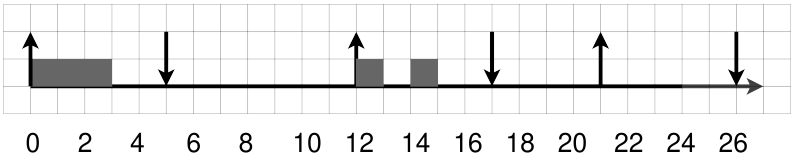
This type of tasks is a middle-ground between aperiodic and periodic tasks. Sporadic tasks are aperiodic tasks characterised by a *minimum inter-arrival time* between jobs.
In this case we have a generic structure:

```
1 void *PeriodicTask(void *arg){
2   //Initialization
3   while (cond) {
4     //Computation
5     //Wait for an event
6   }
7 }
```

From a mathematical model point of view, a sporadic task can be formalized with:

$$\begin{aligned} r_{i,k+1} \geq & \quad r_{i,k} + T_i \\ d_{i,k} = & \quad r_{i,k} + D_i \\ C_i = & \quad \max_k(c_{i,k}) \end{aligned}$$

So as a periodic task, also a sporadic task can be described with a triplet $\tau_i = (C_i, D_i, T_i)$, but instead of using a period, T_i is the Minimum Inter-arrival Time (MIT). Moreover notice that the next activation time $r_{i,k+1}$ is *greater than or equal to* the previous activation task $r_{i,k}$ plus the minimum inter-arrival time T_i, hence two jobs cannot be activated with a smaller delay than T_i.
An example of sporadic tasks is the following:



2.1.5 Task Criticality

A deadline is said to be **hard** if the missing of such deadline causes a critical failure in the system.

Definition 2.1: Hard Real-Time Task

A task is said to be a *hard real-time* task if all its deadlines are hard:

$$\forall j, \rho_{i,j} \leq D_i \Rightarrow R_i \leq D_i$$

This means that before starting the task, all the deadlines must be guaranteed.
There is also the opposite kind of tasks: such tasks are said to be *soft real-time* tasks and they depend on *soft* deadlines. In these cases a deadline miss does not mean a failure in the system, but simply a degradation in the *quality of service*.
With soft real-time tasks, some deadlines can be missed, but the more deadlines are missed, the more the service degrades. We need to express the quality of service in some way, for example we could ask to not have more than X consecutive deadlines misses, to not miss deadlines for a certain interval of time T, or that the number of missed deadlines must be below a certain threshold. These examples are still *deterministic*.
There are also other criteria which deal with probability, for example we don't want the probability of a deadline miss to be over a certain value. Another possibility is to use a *deadline miss ratio* instead of using a probability:

$$\frac{\#Missed\ deadline}{\#Deadline} \leq R_{max}$$

2.1.5.1 Examples of Soft Real-Time Tasks

Let's consider a video player which should maintain a value of $25fps$ (frame per second) which means having a frame period of $40ms$.
It can happen that a frame is delayed or must be skipped, but as long as the number of frames skipped is under a certain value, the user will not see any degradation.
In some robotic system, some actuations can be delayed with little consequences on the control quality. For example, suppose that with a given period, the control loop should check the sensors and make computation, it's possible to show that the system can survive until a certain level of missed control checks.
Notice that soft tasks do not mean no guarantees, instead these should always be present in real-time systems.

2.1.6 Job Execution Time

We have said that tasks can have variable execution times between different jobs, e.g., this could be due to different inputs.
Other causes of different execution times are:

- Hardware issues, for example cache effects or pipeline stalls;
- The internal state of the stack;

2.2 Concurrent Tasks – Scheduling

Task don't run on bare hardware, but they need an operating system which creates the illusion of having more CPUs so that multiple tasks can be executed in parallel. The part of the operating system that allow this "magic" is the kernel.
Concurrency is implemented by multiplexing tasks on the same CPU, that is tasks are alternated on a single real processor. In order to alternate them correctly a **task scheduler** decides which task executes at a given instant in time. The goal of the scheduled is to allocate the tasks in such a way that they respect the various temporal constraints, i.e., their deadlined. A scheduler is simply a component that generated a schedule from a set of tasks.

Definition 2.2: Schedule

A schedule is a function $\sigma(t)$ that maps time t into an executing task:

$$\sigma : t \to \mathcal{T} \cup \tau_{idle}$$

where $\mathcal{T}$ is the task-set and τ_{idle} is the *idle task*.

The idle task is an abstraction to say that if the system doesn't have any task to execute, then it will do nothing.
So basically a schedule is the previous defined function and the scheduler is the component that implements said function.
We can extend this notion to Symmetric Multi-Processing (SMP) systems: consider m equal CPUs, then $\sigma(t)$ can be extended to map t in vectors $\tau \in (\mathcal{T} \cup \tau_{idle})^m$.

Definition 2.3: SMP Systems Schedule

A schedule is a function $\sigma(t)$ that maps time t into a vector of executing task which length is the number of multiprocessors m:

$$\sigma : t \to (\mathcal{T} \cup \tau_{idle})^m$$

From an algorithmic point of view, a *scheduling algorithm* is used to select for each time instant t a task to be executed on a CPU among the ready tasks. As it's possible to grasp, there are more scheduling algorithms and they may create different schedules. Said $\mathcal{A}$ the chosen algorithm, it generates the schedule $\sigma_{\mathcal{A}}(t)$.

Definition 2.4: Schedulable Task

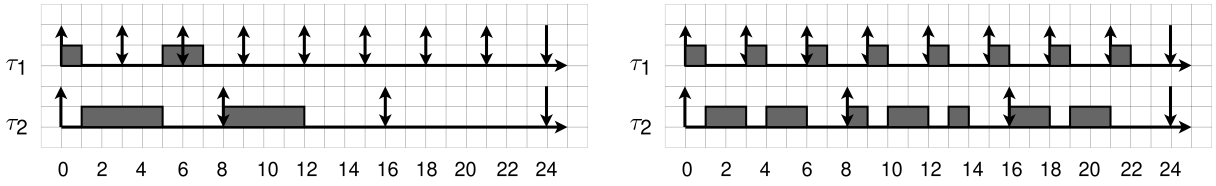
We say that a task set is schedulable by an algorithm $\mathcal{A}$ if $\sigma_{\mathcal{A}}$ does not contain missed deadlines.

From this then we can construct **schedulability tests** to check if a given task-set $\mathcal{T}$ is schedulable by a given algorithm $\mathcal{A}$.
Consider a task set composed of two periodic tasks for which the deadlines correspond to the period of the tasks:

$$\mathcal{T} = \{(1, 3), (4, 8)\}$$

It's possible to see that a First Come First Served (FCFS) algorithm cannot schedule the task-set correctly. Indeed by looking at Figure 2.1a it's possible to see that τ_2 has taken up all the

processing space for τ_1.
In Figure 2.1b, an algorithm which consists of first scheduling τ_1 and then τ_2, that is giving priority to τ_1, is used and it's possible to see that the task set is schedulable. Notice that if we



(a) First Come First Served (FCFS)

(b) $\mathcal{A}$ gives priority to τ_1

Figure 2.1: Example of possible scheduling algorithm. On the left a FCFS approach that does not work with the task set, while the proposed algorithm on the right is able to schedule correctly the task set.

were to give a higher priority to τ_2, there would have been a violation since τ_1 would not execute some times.

2.2.1 Cyclic Execution Scheduling

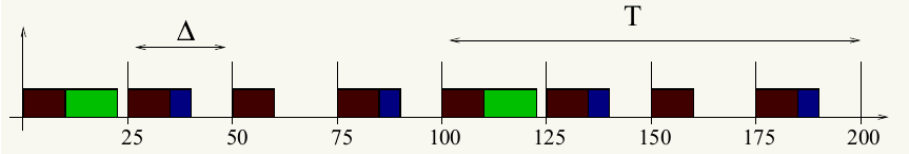
In some cases, we can compute the schedule of the tasks off-line, that is before executing the tasks. In these cases there isn't even the need of an operating system, such is its simplicity, and it's well-tested since it's used a lot, for example in air traffic control which needs to be sure that the scheduling will be possible.
Cyclic execution scheduling was originally thought for periodic tasks but can be extended to aperiodic tasks.
The idea is to have a static scheduling algorithm, that is the schedule is decided once and then never changed. Another fundamental aspect is that the jobs are *not preemptable*: one a job takes the CPU, it executes until termination.
What we are going to do is to divide our time into slots and then allocate each slot statically to a task.
A periodic timer activates the execution of the jobs. There are two cycles:

- **Major Cycle**: it is the least common multiple of all the tasks' periods;
- **Minor cycle**: it is the greatest common divisor of all the tasks' periods.

Our timer fires every minor cycle Δ.

2.2.1.1 Example

Consider the following task-set: $\mathcal{T} = \{\tau_1, \tau_2, \tau_3\}$, where each task $\tau_i = (C_i, D_i, T_i)$ has the deadline D_i set equal to the period T_i which are: $T_1 = 25ms,\ T_2 = 50ms,\ T_3 = 100ms$.
The minor cycle then is: $\Delta = gcd(25, 50, 100) = 25ms$, while the major cycle is $T = lcm(25, 50, 100) = 100ms$.
Let's compute a schedule that respects the tasks' periods. We are going to allocate slots of size $\Delta = 25ms$ and the schedule will repeat every $T = 100ms$. The only constraint is that in every minor cycle, the tasks must execute for less than $25ms$.



The advantages of this scheduling are:

- It is a simple implementation: no real-time operating system is required, we just need a single stack for all the tasks.
- Non-preemptable scheduling, so no need to protect the data with different techniques which would also slow down the computations.
- Much easier to verify that the scheduling is correct and provide formal proof of the correctness of the code.
- Low run-time overhead.
- Jitter can be explicitly controlled.

On the contrary the disadvantages are:

- Not being preemptive, if the tasks were to take more time than what they are supposed to, the schedule would break. So it is not robust during overloads. One solution would be to analyze all the code and break it down in smaller pieces and then reschedule using said pieces.
- Since its a static schedule it's difficult to expand it if new tasks are introduced since all the schedule should be recomputed.
- Not easy to handle aperiodic/sporadic tasks since we have to reserve a space every time wasting time if the task does not execute in that cycle.
- All tasks' periods must be a multiple of the minor cycle time.
- Difficult to incorporate processes with long periods.

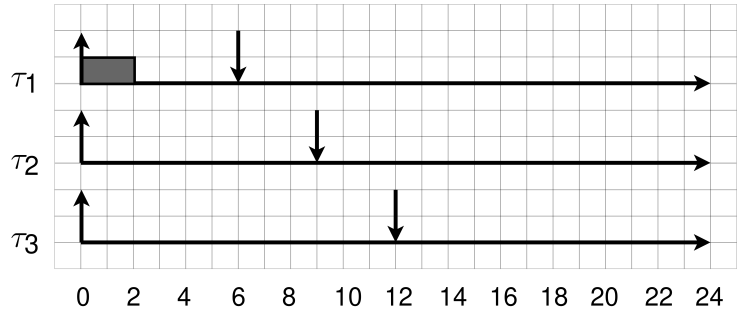
2.2.2 Fixed Priority Scheduling

An alternative to the previous algorithm is *fixed priority scheduling*. In this case, we need to have an operating system and we'll have to deal with concurrency. Indeed, this is a *preemptive* scheduling algorithm, which means that the tasks can be stopped by the execution of other tasks with higher priority. A *fixed priority* p_i is assigned, as an integer value, to every task τ_i and the active task is chosen as the one with the highest priority[1].
Notice that in this case the priorities are fixed, in the sense that they are assigned once and never changed.

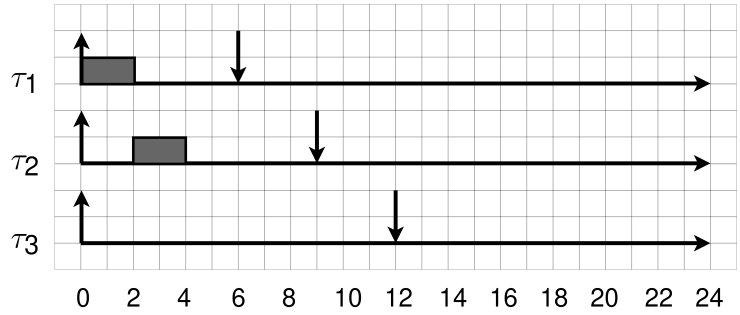
2.2.2.1 Example 1

Consider the following task-set: $\mathcal{T} = \{\tau_1 = (2, 6, 6), \tau_2 = (2, 9, 9), \tau_3 = (3, 12, 12)\}$ with priorities $p_1 = 3,\ p_2 = 2,\ p_3 = 1$, respectively.
Let's say that the three tasks start at the same time. Since τ_1 has the highest priority, it is the first one to be scheduled.

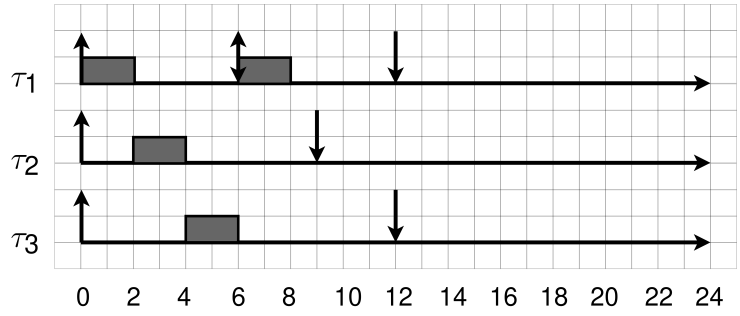
[1] In the literature, the lowest number is the one with the highest priority.



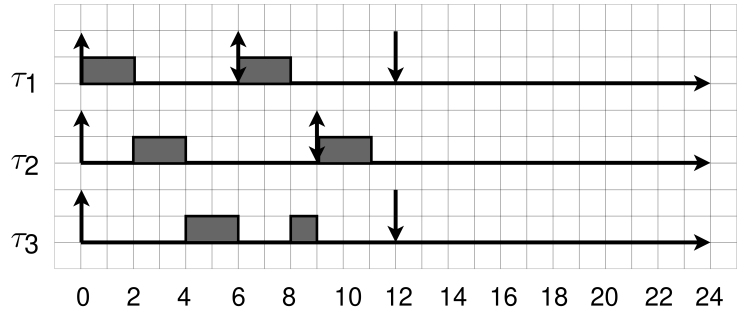
Then the second process τ_2 is the one which has the highest priority, hence it will be executed before τ_3:



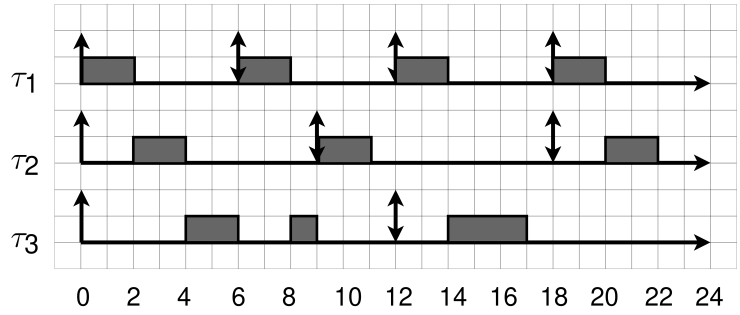
Finally also the τ_3 can start to execute, but at time $t = 6$ it has to stop because τ_1 is asking again for the CPU and it has a higher priority than τ_3:



When τ_1 has finished, then τ_3 can complete its execution right before τ_2 starts again.

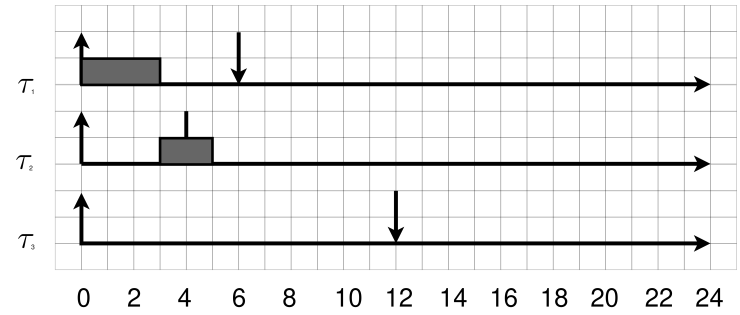


By completing to $t = 24$:



2.2.2.2 Example 2

Consider the following task-set: $\mathcal{T} = \{\tau_1 = (3, 6, 6), \tau_2 = (2, 4, 8), \tau_3 = (2, 12, 12)\}$ with priorities $p_1 = 3,\ p_2 = 2,\ p_3 = 1$, respectively.
Let's suppose that the tasks starts all together. Then τ_1 is the one with the highest priority, hence it's the first to be scheduled. After completing, tasks τ_2 can be scheduled, but at this point it will not make the deadline.



Such task-set is not schedulable with this algorithm. We can ask ourselves whether the problem is the choice of the priorities, or if the whole model is not suit for our case. In the first case there is a solution, i.e., changing priorities, while for the second case, one should changed the scheduling model.

2.2.3 Assigning Priorities

To use fixed priority scheduling one should consider:

- The response time of the task with the highest priority should be minimum and it should be equal to its Worst-Case Execution Time (WCET);
- The response time of the other tasks depends on the interference of the higher priority tasks, for instance, in example of Section 2.2.2.1, τ_1 having the highest priority always block τ_2 since it takes control whenever it want;
- The priority assignment may influence the schedulability of a task-set.

The question then is how to assign priorities. There are two possible policies:

- **Schedulability**: finding the best priorities means making all tasks schedulable;
- **Response time**: finding the best priorities means minimizing the response time of a subset of tasks.

We will focus on the first policy.
An optimal priority assignment *Opt* is such that:

- If the task set is schedulable with another priority assignment, then it is schedulable with priority assignment *Opt*.
- If the task set is not schedulable with *Opt*, then it is not schedulable by any other assignment.

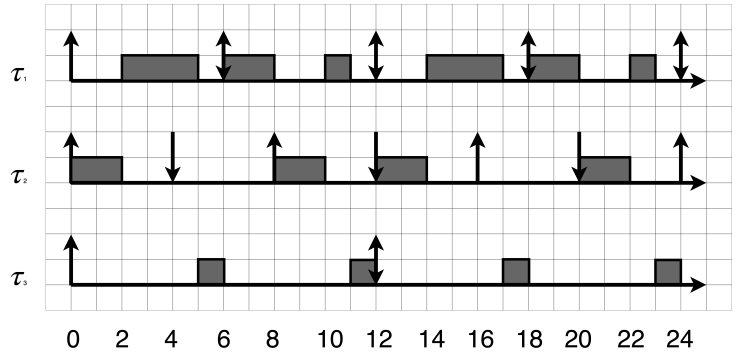
That is there is no other priority that produces a feasible schedule while the optimal one does not.

2.2.3.1 Rate monotonic (RM) Assignment

Consider a periodic task-set T with all tasks having relative deadline D_i equal to the period T_i, and with all offsets equal to 0, $(\forall i, r_{i,0} = 0)$.
It can be proven that the best assignment is the **Rate Monotonic** assignment, which states that if the priorities are fixed, then the optimal solution is achieve by giving the highest priorities to the tasks with the shortest activation time. If a task-set cannot be scheduled with rate monotonic when the above conditions are met, then it cannot be scheduled with fixed priorities.

2.2.3.2 Deadline monotonic (DM) Assignment

Given a periodic task-set with deadlines different from periods and with all offset equal to 0, $\forall i, r_{i,0} = 0$, then the best assignment is the **Deadline monotonic** one, which states that the highest priority should be given to the task with the shortest *relative* deadline, i.e., to the task with the least time remaining to complete before the deadline.
Notice that DM assignment uses relative deadlines and not the absolute ones: let's say that we have two tasks τ_1 and τ_2, for which the absolute deadline is equal to the period. Then if we choose priorities w.r.t. the absolute deadline, when the first deadline has ended, the other process has the nearest absolute deadline and the priority should be changed, but we are talking about fixed priorities. With relative deadlines, the assignment of priorities is fixed (keep in mind that this are periodic tasks).
These are not the only assignments, but they are the optimal for fixed priority scheduling.
Consider the example from Section 2.2.2.2, which had the following task-set: $\mathcal{T} = \{\tau_1 = (3, 6, 6),\ \tau_2 = (2, 4, 8),\ \tau_3 = (2, 12, 12)\}$. We have seen that the priorities $p_1 = 3,\ p_2 = 2,\ p_3 = 1$ do not work. Let's try to apply the deadline monotonic assignment, that is, assigning the following priorities: $p_1 = 2, p_2 = 3, p_3 = 1$:



It's possible to see that each $t = 12$ time units, all the task synchronize back, hence if in the first 12 units of time there were no problems, then also afterwards there should not be. Then

we see that this assignment works and we know it's the optimal.

2.2.4 Schedulability Test

Given a task-set, how can we guarantee that it is schedulable or not?
The first possibility is to simulate the system to check that no deadline is missed. In this case, we set the computation time of a task to its Worst-Case Execution Time. If while executing a deadline miss was found, then we are in violation and there is no way to schedule such task-set since the optimal assignment algorithms fail. The question though is: how long do we have to continue the simulation before safely saying there is no violation?
This is trivial with periodic tasks with no offset, i.e., they all start together, since it is sufficient to simulate the schedule until the *hyperperiod*, that is the least common multiple of the tasks period: $H = lcm(T_i)$. If no violation happens in this period of time, then it will never happen. When we consider task-set with offsets ϕ_i, then we need to simulate until $2H + \phi_{max}$.
Notice that, if task periods are prime numbers, then the hyperperiod can be very large.

Definition 2.5: Jitter

Jitter is defined as the amount of time a function takes to give an output. We shall differentiate between:

- Input jitter: the maximum difference between activation times;
- Output jitter: the maximum difference between releasing times.

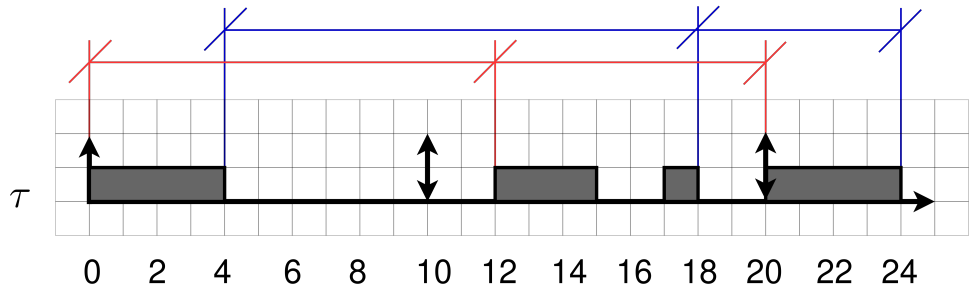


Figure 2.2: In red the input jitter, which is 12 units of time, while in blue the output jitter which is 14 units of time.

When considering sporadic tasks, we can assume them to arrive at the highest possible rate, so we fall back to the case of periodic tasks with no offset. If we have a minimum inter-arrival time and we want to do a worst-case scenario, then this coincides when the sporadic tasks are back to back with one another.

2.2.4.1 Utilization Bound

In some cases it is useful to have a very simple test to see if the task-set is schedulable. A sufficient[2] test is based on the **utilization bound**.

[2] A sufficient test means that if it passes, then the hypothesis is correct, while if it fails nothing can really be said.

Definition 2.6: Utilization Least Upper Bound

The utilization least upper bound for a scheduling algorithm $\mathcal{A}$ is the smallest possible utilization U_{lub} such that for any task-set $\mathcal{T}$, if the task-set's utilisation U is less than or equal to U_{lub}, then the task-set is schedulable by algorithm $\mathcal{A}$.

$$U \leq U_{lub}$$

Each task uses the processor for a fraction of time:

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

where T_i is the task period and C_i is the WCET. The total processor utilisation is:

$$U = \sum_i \frac{C_i}{T_i}$$

This is a measure of the processor's load. We don't want the utilization to be too low because then we are overpaying for the machine, but also we want some margin to be safe.
If we are using rate monotonic assignment (Section 2.2.3.1), then, given n periodic tasks with relative deadline equal to periods, it's possible to prove that the utilization least upper bound is given by:

$$U_{lub} = n\left(2^{\frac{1}{n}} - 1\right)$$

where n is the number of tasks. Also for large n we have: $U_{lub} \approx 0.69$. It's possible to notice that it is a decreasing function of n, so the more tasks we have, the less performing it becomes since the maximum utilization decreases.
It's possible to notice that if $U > 1$, then the task-set is for sure not schedulable, because it would mean that the tasks take more time than what the processor is giving. Having $U < 1$ is obviously a necessary condition. Since we know that $U < U_{lub}$, then we'd like to have $U_{lub} \to 1$, in this way we'd have a necessary and sufficient condition.
Remember that if $U_{lub} < U \leq 1$, then the task-set may or may not be schedulable.
Therefore the schedulability test consists in:

- Computing $U = \sum_{i=1}^{n} \frac{C_i}{T_i}$
- If $U \leq U_{lub}$ the task set is schedulable;
- If $U > 1$ the task set is not schedulable;
- If $U_{lub} < U < 1$ the task set may or may not be schedulable.

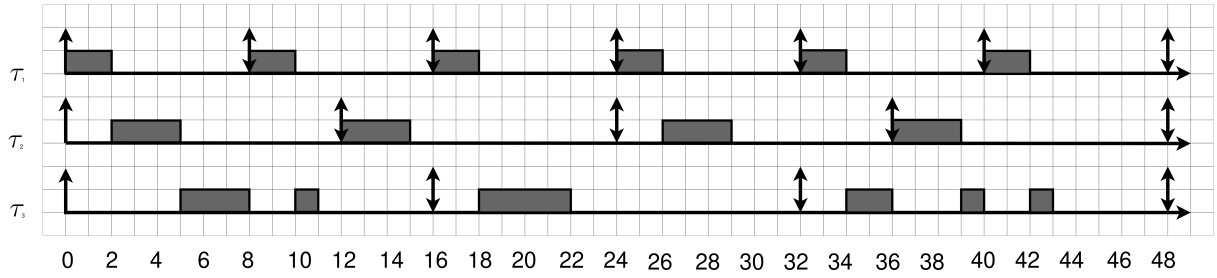
2.2.4.1.1 Example 1

Consider a task set $\mathcal{T} = \{\tau_1 = (2, 8), \tau_2 = (3, 12), \tau_3 = (4, 16)\}$ and an utilization least upper bound of 77%. Say whether it is schedulable or not and if it is, find the priorities and show a time representation.
First of all we should compute the usage of the system:

$$U = \sum_i U_i, \quad U_i = \frac{C_i}{T_i}$$

τ_i	U_i
τ_1	$^2/_8 = 0.25$
τ_2	$^3/_{12} = 0.25$
τ_3	$^4/_{16} = 0.25$
U	0.75

Since the total utilization is less than U_{lub}, then we can say that the task-set is schedulable.
From the moment that the tasks' periods are the same as the deadlines, then the optimal assignment is given by rate monotonic assignment (Section 2.2.3.1), hence we are giving the following priorities: $p_1 = 3,\ p_2 = 2,\ p_3 = 1$:



In this case we didn't need to show the full graph since it was already proven.

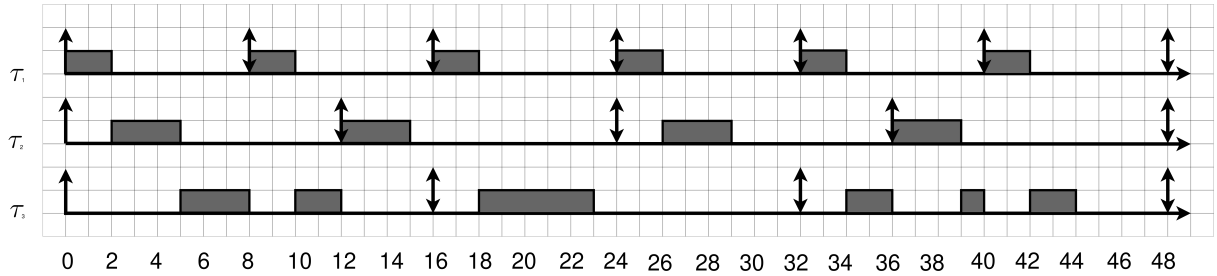
2.2.4.1.2 Example 2

Consider a task set $\mathcal{T} = \{\tau_1 = (2, 8), \tau_2 = (3, 12), \tau_3 = (5, 16)\}$ and an utilization least upper bound of 77%. Say whether it is schedulable or not and if it is, find the priorities and show a time representation.
First of all we should compute the usage of the system:

$$U = \sum_i U_i, \quad U_i = \frac{C_i}{T_i}$$

τ_i	U_i
τ_1	$^2/_8 = 0.25$
τ_2	$^3/_{12} = 0.25$
τ_3	$^5/_{16} = 0.3125$
U	0.81

In this case we have $U_{lub} < U \leq 1$, hence it's not possible to say a priori whether this is schedulable or not, but we actually need to do the temporal graph. The least common multiple for the durations is $lcm(8, 12, 16) = 48$.



Let's now see how to deal with *Deadline Monotonic assignment* (Section 2.2.3.2): instead of considering periods T_i, we are going to consider the deadlines D_i:

$$U' = \sum_{i=1}^{n} \frac{C_i}{D_i}$$

The idea is to consider $\tau = (C, D, T)$ as $\tau' = (C, D, D)$; then, τ' is a "worst case" for τ and so if τ' can be guaranteed, then so can τ. From this point, the test is simply the same as before.
That said, for deadline monotonic assignment, this value is rarely used since this is a worse estimator of the utilization, so much so that if the utilization U' is greater than 1, it could still be correct.
Moreover in general this bound is very pessimistic, most of the times a task-set with $U > U_{lub}$ is schedulable anyway.
A particular case is when tasks have periods that are harmonic, that is for every two tasks τ_i, τ_j, either T_i is multiple of T_j or vice versa. In this case, the utilization bound is $U_{lub} = 1$ and rate monotonic is an optimal algorithm for harmonic task-sets.

2.3 Real-Time Scheduling Analysis

When the choice for assigning priorities to the tasks is not rate monotonic and when we have $U_{lub} < U \leq 1$, the previous test fails.

2.3.1 Worst-Case Execution Time Analysis

A necessary and sufficient test it to compute the worst-case response time R_i for every task. We know that if $R_i \leq D_i$, that is, the worst-case response time is less than the deadline, then we know that the task schedulable since we are considering the worst possible outcome. To prove the task-set's schedulability, we have to prove that for all the tasks in the task-set, all the worst-case response times are less then their deadlines. In this way we are actually not making any assumption on the priority assignment.
There are though some assumptions we are going to make are:

- All periodic tasks are without offset;
- There is no sporadic task: as seen before, they can be reduced to period tasks.

We are going to use decreasing priority notation: $i < j \rightarrow p_i > p_j$.

Theorem 2.1: Critical Instant

The critical instant for task τ_i occurs when job $J_{i,j}$ is released (deadline) and is requested at the same time with requests for also higher priority tasks.

This is due to the fact that it is the worst possible offsets combination. A job $J_{i,j}$ released at the critical instant experiences the maximum response time for $\tau_i : \forall k, \rho_{i,j} \geq \rho_{i,k}$. [3]
Notice that, if every offset is 0, then the first job of every task is released at the critical instant. So it's possible to say that the worst response time R_i for task τ_i depends on:

- It's execution time;

[3] Note that this is a simplified version since also jobs' deadlines should be considered.

- The execution time of higher priority tasks since these can preempt task τ_i and increase its response time.

Hence the response time R_i can be defined as:

$$R_i = C_i + \sum_{h=1}^{i-1} \left\lceil \frac{R_i}{T_h} \right\rceil C_h$$

So the response time R_i is actually a function of itself: $R_i = f(R_i)$. Since there is no closed-form expression for computing the worst case response time R_i, the only way to solve this problem is using an iterative method until convergence is reached.
Let $R_i^{(k)}$ be the worst case response time for τ_i at step k. The correct solution for R_i is given by:

$$R_i = \lim_{k \to \infty} R_i^{(k)}$$

The first guess we could do is simply say that the worst response time corresponds to the computational time C_i:

$$R_i^{(0)} = C_i$$

Then a better guess can be made by also considering the fact that before running this job, the ones with a higher priority must have been executed, hence:

$$R_i^{(0)} = C_i + \sum_{h=1}^{i-1} C_h$$

Then the k-th result is computed as:

$$R_i^{(k)} = C_i + \sum_{h=1}^{i-1} \left\lceil \frac{R_i^{(k-1)}}{T_h} \right\rceil C_h$$

The algorithm stops in two situations:

- Either this is actually the worst response time (it has converged):

$$R_i^{(k+1)} = R_i^{(k)}$$

- Or the task is not schedulable

$$R_i^{(k)} > D_i$$

The iterative approach is actually a standard method to solve non-linear equations: either the solution exists and it converges to it, or a certain condition (in this case $R_i^{(k)} > D_i$) avoids infinite iterations. [4]
This algorithm is also known as worst-case execution time analysis and in the worst case the number of steps required for convergence N is actually exponential in the number of tasks:

$$N \propto \sum_{h=1}^{i-1} \left\lceil \frac{D_h}{T_h} \right\rceil$$

Such worst case is pretty rare: usually the number of steps is low.
Let's consider a task and we carry out the worst case response time analysis. After we have

[4] If something feels off:
`vlc -start-time 892 $HOME/GoogleDrive/Magistrale/5sem/RTOSM/Lectures/RTOS_09_24_2020_3.mp4`

finished the computations we end up with a task which can allegedly be feasibly schedulable. If we look at the difference between the relative deadline and the response time computed right now, we may think that this is a measure of the robustness of the schedule, but we would be wrong. Indeed, if we look more carefully, we can see that there are cases in which by modifying the computation time, all of a sudden we get other activations for the priority tasks. This means that it is highly discontinuous, that is by changing a tiny bit we could have large different results. Therefore we cannot make any assumption about the robustness of the schedule by comparing the worst case scenario and the relative deadline.

2.3.2 Example 1

Consider the following task set: $\{\tau_1 = (2,5), \tau_2 = (2,9), \tau_3 = (5,20)\}$ and an utilization $U = 0.872$. The easiest thing is to consider τ_1 since it is the task with the highest priority and so its execution time corresponds to its response time. In case of the other two tasks, we need to consider the interference of the other tasks. We shall concentrate on τ_3.
We start by making our guess given by the sum of the execution times:

$$R_3^{(0)} = C_1 + 1 \cdot C_1 + 1 \cdot C_2 = 9$$

The question then is: did I converge? To see this we need to put the value inside the following formula:

$$R_i^{(k)} = C_i \sum_{h=1}^{i-1} \left\lceil \frac{R_i^{(k-1)}}{T_h} \right\rceil C_h$$

$$R_3^{(1)} = C_3 + \left\lceil \frac{9}{5} \right\rceil \cdot C_1 + \left\lceil \frac{9}{9} \right\rceil \cdot C_2 = C_3 + 2 \cdot C_1 + C_2$$

So in the second iteration we need to consider 2 times C_1. The next guess is 11 and we try to compute a new iteration:

$$R_3^{(2)} = C_3 + \left\lceil \frac{11}{5} \right\rceil \cdot C_1 + \left\lceil \frac{11}{9} \right\rceil \cdot C_2 = C_3 + 3 \cdot C_1 + 2 \cdot C_2 = 15$$

So also in this case we haven't converged and we can repeat again with 15:

$$R_3^{(3)} = C_3 \left\lceil \frac{15}{5} \right\rceil \cdot C_1 + \left\lceil \frac{15}{9} \right\rceil \cdot C_2 = C_3 + 3C_1 + 2C_2 = 15$$

So in this case we have converged.

2.3.3 Example 2

Let's now consider the following case in which the deadlines do not correspond to the periods:

$$\tau_1 = (1,4,4), p_1 = 3, \quad \tau_2 = (4,6,15), p_2 = 2, \quad \tau_3 = (3,10,10), p_3 = 1; \quad U = 0.72$$

We can start by trying the formula used before:

$$R_i^{(k)} = C_i \sum_{h=1}^{i-1} \left\lceil \frac{R_i^{(k-1)}}{T_h} \right\rceil C_h$$

$$R_3^{(0)} = C_3 + C_1 + C_2 = 8$$

$$R_3^{(1)} = C_3 + \left\lceil \frac{8}{15} \right\rceil C_2 + \left\lceil \frac{8}{4} \right\rceil C_1 = 9$$

$$R_3^{(2)} = C_3 + \left\lceil \frac{9}{15} \right\rceil C_2 + \left\lceil \frac{9}{4} \right\rceil C_1 = 11$$

$$R_3^{(3)} = C_3 + \left\lceil \frac{11}{15} \right\rceil C_2 + \left\lceil \frac{11}{4} \right\rceil C_1 = 11$$

So we have found convergence.

2.3.4 Time Demand Analysis

We want to see if a task is schedulable. Let's consider any interval of time between 0 and the deadline, and compute how much computing time our task demands in that interval and if such requirement is matched by the system offer, then our system is fine.
Basically: in any interval, the computation demanded by all the tasks in the set must never exceed the available time.
The problem is how to compute the time demanded by a task set $\mathcal{T}$.
Given an interval $[t_1, t_2]$, let $\mathcal{J}_{t_1,t_2}$ be the set of jobs that started after t_1 and with deadline lower than or equal to t_2:

$$\mathcal{J}_{t_1,t_2} = \{J_{i,j} : r_{i,j} \geq t_1 \wedge d_{i,j} \leq t_2\}$$

The processor demand in $[t_1, t_2]$ is defined as:

$$W(t_1, t_2) = \sum_{J_{i,j} \in \mathcal{J}_{t_1,t_2}} c_{i,j}$$

The worst case scenario is C_i instead of $c_{i,j}$.
This means that the time demand analysis requires that:

$$\forall (t_1, t_2), W(t_1, t_2) \leq t_2 - t_1 \tag{2.1}$$

So we need to check all the possible t_1, t_2 combinations in a hyperperiod. This is obviously highly inefficient.
If we want to verify the schedulability of a task, we need to check only its first job and forget about the other job activations. So the processor demand $W(t_1, t_2)$ is actually the time demanded in $[t_1, t_2]$ by all the tasks τ_j with $p_j \geq p_i (\Rightarrow j \leq i)$. Basically the processor demand of a task can be computed by taking into account only the first job of the task and all of the tasks with a schedulability priority higher than the considered tasks.
So instead of computing all the possible combinations of tasks, we can check one task at a time. Moreover we can consider only the intervals that start from the critical instant and ending before or on the deadline of the task: $W_i(0, t), 0 \leq t \leq D_i$ We need to sum up all the jobs that fit between 0 and t, with $t \in [0, D_i]$. The value is computed as:

$$W_i(0, t) = C_i + \sum_{h=1}^{i-1} \left\lceil \frac{t}{T_h} \right\rceil C_h$$

Which is the worst case of my task plus the worst cases of all the tasks with higher priority than my task.
We use the ceiling because in this way, instead of asking for all the possible intervals t_1, t_2 (Equation 2.1), we can ask if there exists an instant of time in which our demand is met by the offer. This means a drastic simplification since we don't have to check all the possible intervals, but we need to find just one that satisfies the condition.

Theorem 2.2

A task τ_i is schedulable if and only if

$$\exists t : 0 \leq t \leq D_i \wedge W_i(0,t) \leq t$$

A task set $\mathcal{T}$ is schedulable if and only if

$$\forall \tau_i \in \mathcal{T}, \exists t : 0 \leq t \leq D_i \wedge W_i(0,t) \leq t$$

2.3.4.1 Example 1

Let's consider the following example:

$$\tau_1 = (c_1, 10) \quad \tau_2 = (c_2, 20) \quad P_1 > P_2$$

If we consider the first task τ_1 and want to see what is its time demand. Since it has the highest priority, the time demands is given simply by its computation time:

$$W_1(0,t) = c_1$$

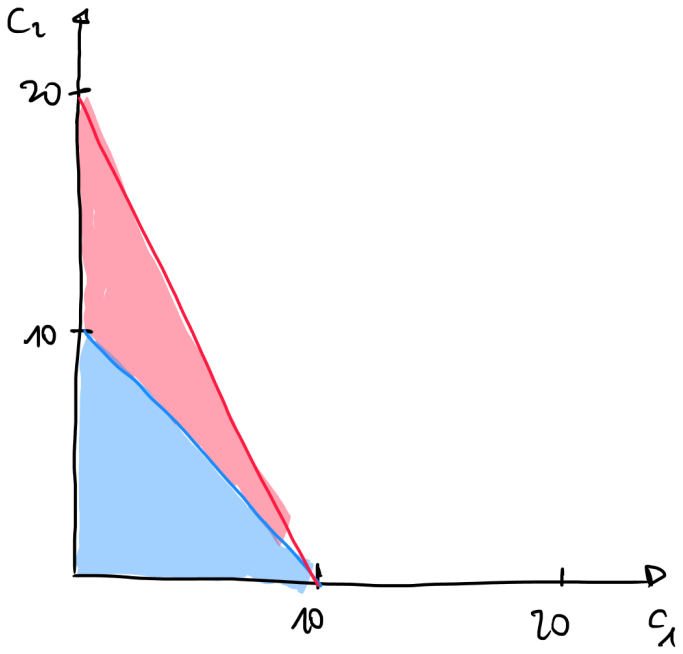
This means that we simply need to find a moment in time, before t, for which the computing time is higher than or equal to this. If such thing can be found, then the task is schedulable.
For example, if $t = 10$, then we would have $W_1(0,10) \leq 10$, but we know that $W_1(0,10) = c_1$, so we have that $c_1 \leq 10$, which is pretty straightforward since τ_1 is the task with the highest priority, hence it does not get interrupted by other tasks and the only constraint is that it must finish before its own deadline.
Now let's consider τ_2. In this case, τ_1 has a higher priority so we need to take also that task into consideration:

$$W_2(0,t) = c_2 + \left\lceil \frac{t}{T_1} \right\rceil c_1 = c_2 + c_1 \left\lceil \frac{t}{10} \right\rceil$$

We need then to check two cases:

- $t = 10$ in which case it must be: $c_1 + c_2 \leq 10$;
- $t = 20$ in which case it must be $2c_1 + c_2 \leq 20$.

We need just one of the two to be true since we have changed the $\forall$ with an $\exists$.
It's possible to notice that the above means the following graphics:



The two triangles show the possible values of c_1, c_2 which would satisfy the requirements. The goal is to find a value inside the union of the areas, which basically is finding a value inside the big red rectangle (which obviously spans also under the blue one).
Moreover, once we have a valid point, we can compute the distance from the border, i.e. the hypotenuse, to have a measure of robustness.

The theorem can also be defined by introducing the ration L between demand and offer:

$$L_i(t_1, t_2) = \frac{W_i(t_1, t_2)}{t_2 - t_1}; \quad L_i = \min_{0 \leq t \leq D_i} L_i(0, t); \quad L = \max_{\tau_i \in \mathcal{T}} L_i$$

So a task τ_i is schedulable if and only if $L_i \leq 1$ and the task set $\mathcal{T}$ is schedulable if and only if all the tasks inside $\mathcal{T}$ are schedulable, i.e., the minimum of each task is less than 1.
Notice that we don't have to look over all the possible points in the interval, but just on the break points, that is when the processor demand make a step up.
This is to say that is sufficient to consider the scheduling points which are the multiple of the period, compute the value of the demand in this points and see if they are smaller than 1.

2.3.4.2 Example 2

Let's consider the following three tasks ordered by priority:

$$\tau_1 = (20, 100), \tau_2 = (40, 150), \tau_3 = (100, 350)$$

We know that τ_1 is schedulable if $c_1 = 20 < 100$, which is always the case and we don't have problems.
Know let's consider τ_2. In this case we need to consider also τ_1 since it has a higher priority and also we need to consider two schedulable points:

- $t = 100 \rightarrow$

$$W_2(0, 100) = c_2 + c1 \left\lceil \frac{t}{T_1} \right\rceil = 40 + 20 = 60 \leq 100$$

In this case τ_2 is schedulable without problems.

- $t = 150 \rightarrow$

$$W_2(0, 150) = c_2 + c1 \left\lceil \frac{t}{T_1} \right\rceil = 40 + 20 \times \lceil 1.5 \rceil = 40 + 2 \times 20 = 80 \leq 150$$

Also in this case τ_2 is schedulable.

Finally consider the final task. In this case, we need to consider the first two tasks, plus five schedulable points:

- $t = 100 \rightarrow$

$$W_3(0, 100) = c_3 + c1 \left\lceil \frac{t}{T_1} \right\rceil + c2 \left\lceil \frac{t}{T_2} \right\rceil = 100 + 20 + 40 = 160 \leq 100$$

In this case τ_3 cannot be schedule, but it was obvious since c_3 alone takes 100 units of time.

- $t = 150 \rightarrow$

$$W_3(0, 150) = c_3 + c1 \left\lceil \frac{t}{T_1} \right\rceil + c2 \left\lceil \frac{t}{T_2} \right\rceil = 100 + 20 * 2 + 40 = 180 \leq 150$$

Also this schedulable point is not fine for τ_3.

- $t = 200 \rightarrow$

$$W_3(0, 200) = c_3 + c1 \left\lceil \frac{t}{T_1} \right\rceil + c2 \left\lceil \frac{t}{T_2} \right\rceil = 100 + 20 * 2 + 40 * 2 = 220 \leq 200$$

Also this schedulable point is not fine for τ_3.

- $t = 300 \rightarrow$

$$W_3(0, 300) = c_3 + c1 \left\lceil \frac{t}{T_1} \right\rceil + c2 \left\lceil \frac{t}{T_2} \right\rceil = 100 + 20 * 3 + 40 * 2 = 240 \leq 300$$

In this case τ_3 is schedulable.

- $t = 350 \rightarrow$

$$W_3(0, 350) = c_3 + c1 \left\lceil \frac{t}{T_1} \right\rceil + c2 \left\lceil \frac{t}{T_2} \right\rceil = 100 + 20 * 4 + 40 * 3 = 300 \leq 350$$

In this case τ_3 is schedulable.

Since at least one of the above is verified, the system is schedulable.
Notice that this was "wasted time" since the task were given fixed monotonic priorities, so we could have known that the tasks were schedulable simply by using the previous formula:

$$\frac{20}{100} + \frac{40}{150} + \frac{100}{350} = 0.753 < 0.779$$

Where 77.9% is the maximum value for the schedulability of three tasks.
If we were to change c_1 to 40, then the previous formula would be false and we would have to run the time demand analysis to be sure whether the tasks are schedulable or not.
If we were to run also response time analysis, we would have found out that also that analysis gave the response that the tasks are schedulable, but this is obvious since they are both necessary and sufficient conditions regarding the same results.
If we were to run also response time analysis, we would have found out that also that analysis gave the response that the tasks are schedulable, but this is obvious since they are both necessary and sufficient conditions regarding the same results.

2.4 Exercises

2.4.1 Exercise 1

Consider the following tasks with their periods and WCETs. Is it possible to find a cyclic schedule?

Task	Period	WCET
A	50	5
B	30	10
C	20	5

First of all we need to find the major cycle, which can be computed as the least common multiplier, that is 300. Then we need to find the minor cycle, which is the greatest common divisor, that is 10. This information tells us that we'll need to make room for 30 slots (each of 10ms).
We can start by positioning the most frequent task, that is task C, which would go in slot 1, 3 and 5. Then we need to find a proper allocation for B which is actually troublesome since, having WCET=10ms takes up a whole slot for itself. Finally A can be placed also in the slots with C since they both take up only half a slot.
It is also possible to set up the cyclic scheduling problem as an Integer Linear Programming (ILP) problem. We want to do this because it allows for better guarantees, for example:

- If there is a solution, it will be able to find it;
- We may have different types of constraints (e.g., jitter or priority);
- We can define a cost function which will be optimal.

We define some variables. Let

- T_N be the number of tasks;
- $\tau_1, \ldots, \tau_N$ be the tasks;
- D_N be the number of minor cycles within a major cycle, i.e., we know already the values of the cycles and we think in terms of allocating the slots.
- T_j be the priority of the task expressed as a multiple of the minimum cycle.
- $J_{i,j}$, be the j-th job (aka activation) for the i-th task, $j \in 1, \ldots, \frac{D_n}{T_j}$.
- D the duration of the minor cycle.

Now we need to find the decision variables. Many choices are available, but one possibility is to define a binary value:

$$y_{i,j,k} = \begin{cases} 1 & \text{if the } j\text{-th job of the } i\text{-th task is executed in the } k\text{-th slot} \\ 0 & \text{otherwise} \end{cases}$$

The constraint is that, for all tasks and for all the jobs, it must allocate one single slot. So, if we sum up all the possible values of y we have 1.

$$\forall \tau_i \in \{\tau_1, \ldots, \tau_N\}, \forall j \in J_{i,\_}, \sum_{h=0}^{D_N} y_{i,j,h} = 1$$

This constraint captures the fact that only one job is put in one slot. Moreover, the total utilization of the slot must not exceed the dimension of the slot:

$$\forall h \in slots, \sum_{i \leq T_N, j \in \tau_i} y_{i,j,h} c_i \leq D$$

where c_i is the worst case execution time.
Another aspect is that the position cannot be random. For example, if we have a job $J_{i,j}$, the job must be place in $[(j-1)T_i,\ jT_i]$ since we want to respect the periodicity of the task. We can express this constraint with:

$$\forall i \leq T_N, \forall j \in \tau_i, \sum_{h \in [(j-1)T_i,\ jT_i]} y_{i,j,h} = 1$$

If we consider two jobs $J_{i,j}, J_{i,j'}$, with $j' > j$, then they have to be allocated in the correct order:

$$J_{i,j}, J_{i,j'}, j' > j, y_{i,j,h} = 1, y_{i,j,h'} = 1 \iff h' > h$$

2.4.2 Exercise 2

Consider the following task set $\mathcal{T}$. Let's compute the worst case response time analysis using Rate monotonic (RM) and Deadline monotonic (DM) priority assignment.

Task	C_i	D_i	T_i
τ_1	1	4	4
τ_2	2	9	9
τ_3	3	6	12
τ_4	3	20	20

The response times are computed with the formula previously used in the examples:

$$R_i^{(k)} = C_i \sum_{h=1}^{i-1} \left\lceil \frac{R_i^{(k-1)}}{T_h} \right\rceil C_h$$

And the value that should be reported is the convergence one.
The priority are set depending on:

- RM: we look at the period times and the smaller they are, the higher the priority will be;
- DM: we look at the deadline times and the smaller they are, the higher the priority will be.

For example, let's consider the rate monotonic assignment. The response time of τ_1 is immediate to compute since it has the highest priority:

$$R(\tau_1) = 1$$

Then we compute the response time of task τ_2, τ_3 and finally of task τ_4:

$$R(\tau_2) = 3 \quad R(\tau_3) = 7 \quad R(\tau_4) = 18$$

Instead, if we used DM, the priority would be: $\tau_1, \tau_3, \tau_2, \tau_4$ and the response times would become:

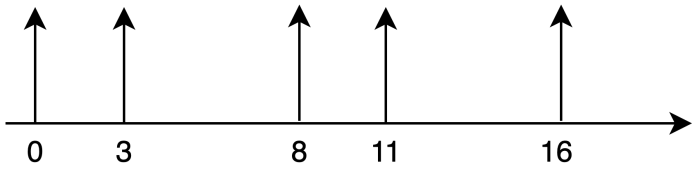
$$R(\tau_1) = 1 \quad R(\tau_3) = 4 \quad R(\tau_2) = 7 \quad R(\tau_4) = 18$$

2.4.3 Exercise 3

Given the non period task τ_1 defined as follows:

- If $j\%2 == 0$ then $r_{i,j} = 8 \cdot \frac{j}{2}$;
- If $j\%2 == 1$ then $r_{i,j} = 3 + 8 \cdot \left\lfloor \frac{j}{2} \right\rfloor$;
- $\forall j, c_{1,j} = 2$;
- The priority of task τ_1 is $p_1 = 3$.

and given the task set $\mathcal{T}$ composed of τ_1 and $\tau_2 = (2, 12, 12)$ and $\tau_3 = (3, 25, 25)$ with $p_2 = 2$ and $p_3 = 1$, compute the worst case response time for τ_2 and τ_3.
Basically the task τ_1 activates at multiples of 8 and multiples of 8 plus 3:



Since τ_1 has the highest priority, its response time is immediate:

$$R(\tau_1) = 2$$

To compute the other response times we first need to take a step back and consider how we got to the formula. For example, the response time for τ_2 is given by its computation time plus all the number of jobs of task τ_1 that can be framed between 0 and the response time of τ_2 times the computation time of τ_1:

$$R(\tau_2) = C_2 + N_{job_{\tau_1}}(R(\tau_2))C_1$$

Usually we compute the number of jobs as the ceiled ratio of the response time over the period, but in this case τ_1 is not periodic. We can find this value for τ_1 by first computing the number of activations for the even part and then adding the number of activations for the odd part. In this last case, we need to "move" the activations down by 3, so to not get into negative numbers we need to consider the max between 0 and the time minus 3:

$$N_{job_{\tau_1}}(t) = \left\lceil \frac{t}{8} \right\rceil + \left\lceil \frac{max(0, t-3)}{8} \right\rceil$$

Generally speaking we have:

$$R_i^{(k)} = C_i + \sum_{h=1}^{i-1} N_{job_{\tau_h}} \left(R_i^{(k-1)} \right) C_h$$

We can then compute the response time of τ_2:

$$R_2^{(0)} = C_2 + C_1 = 4$$

$$R_2^{(1)} = C_2 + N_{job_{\tau_1}}(R_2^{(0)})C_1 = C_2 + \left(\left\lceil \frac{4}{8} \right\rceil + \left\lceil \frac{\max(0,1)}{8} \right\rceil \right) C_1 = 2 + (1+1)2 = 6$$

$$R_2^{(2)} = C_2 + N_{job_{\tau_1}}(R_2^{(1)})C_1 = C_2 + \left(\left\lceil \frac{6}{8} \right\rceil + \left\lceil \frac{\max(0,3)}{8} \right\rceil \right) C_1 = 2 + (1+1)2 = 6$$

So we have converged and $R(\tau_2) = 6$.
we can repeat the same thing with τ_3.

3 Atomicity

An hardware instruction is said to be atomic if it cannot be interleaved with other instructions. For this reason, atomic operations are always sequentialized since they cannot be interrupted. They are commonly used for safe operations, for example, for transferring one word from memory to a register or viceversa.
For example, a variable incrementing may not be an atomic operation, since we first need to load the value, increment it and finally storing it. So an operation such as `x=x+1` may actually be written as:

```
LD R0, x
INC R0
St x, R0
```

Obviously this arises possible inconsistency problems if the operation is interrupted and then resumed. This is the real problem of concurrent programming. Moreover, the problem is not only with operations, but by extension also with tasks. Consider the following two tasks:

```
x=0;
void taskA(){
  x=x+1;
}
void taskB(){
  x=x+1;
}
```

A bad interleaving would be the following one:

```
LD  R0, x  TA
LD  R0, x  TB
INC R0     TB
ST  x, R0  TB
INC R0     TA
ST  x, R0  TA
```

Indeed in this case the final value of `x` will be 1.

3.1 Critical Sections

We say that a shared object where the conflict may happen is a *shared resource*. The part of the code where the problem may happen are called **critical sections**, that is a sequence of operations that cannot be interleaved with other operations on the same resource. Two critical sections on the same resource must be properly sequentialized in order for them to execute in *mutual exclusion* on the same resource.
There are three ways of obtaining mutual exclusion:

- implementing the critical section as an *atomic operation*, that is an operation which is executed at the hardware level, for example when moving a value in memory;
- *disabling preemption system-wide*;
- *selectively disabling the preemption*, for example using semaphores and mutual exclusion.

3.1.1 Atomic Operations

In a single processor system we may as well disable interrupts during the critical section so that even if an interrupt is raised, it will be served when the critical section has finished.
By disabling interrupts, we are basically disabling preemption.
This solution presents some problems:

- If the critical section is long, no interrupt can arrive during the critical section, and so it may interfere with the correct working of the machine. For example, if the machine necessitates a clock being updated every 1ms and the critical section lasts for two, the machine may end up in a bad state.
- Concurrency is disabled during the critical section.

The ideal solution would be trying to avoid conflicts on the resources and not completely disabling the interrupts.

3.1.2 Disabling Preemption

In single processor systems it is possible to disable preemption for a limited interval of time by increasing the priority of the task.
In this case, we are not interfering with the interruptions, but with the execution of the task. This may lead to a problem if a highly priority critical thread needs to execute and it cannot make preemption and it is delayed. Notice that this is a problem especially if the high priority task does not want to access the shared resource.

3.1.3 Selective Disabling Preemption

In this case, we do not want to disable the interrupts for all the tasks, but only for those which need to access the shared resource. In this way, the problem stated before is solved and the most important task can preempt and start its execution when it does not need to access the shared resource.

3.2 Schedulability for Critical Sections

Until now we have considered only independent tasks, meaning that jobs do not block and do not suspend until their termination. In the real world, though, a job might block for various reasons such as:

- Exchange of data through the shared memory (mutual exclusion);
- Synchronization with other tasks while waiting for some data;
- Needing hardware resources which are currently not available.

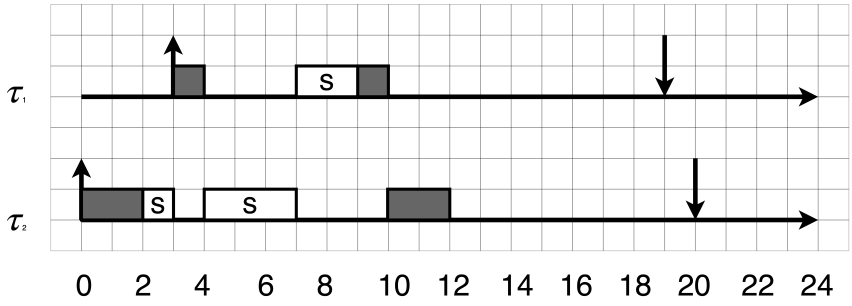
For example, let's consider a control application composed of 3 periodic tasks:

- τ_1 reads the data from the sensors and applies some sort of filters;
- τ_2 reads the filtered data and computes some control law;
- τ_3 reads the outputs and writes on the actuators.

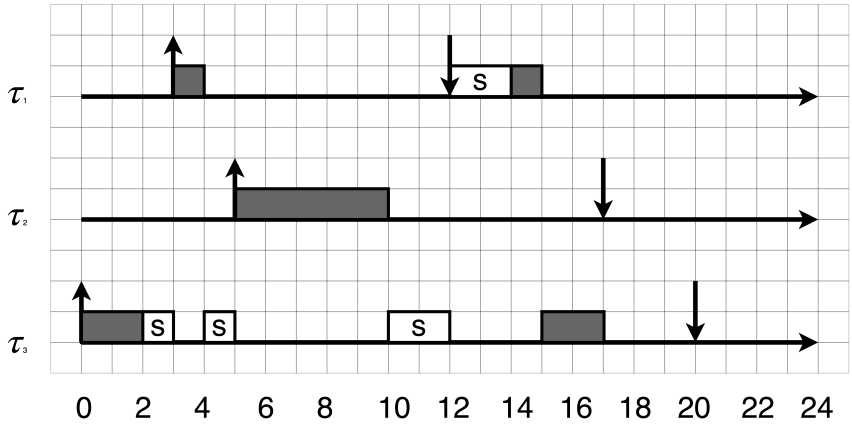
All the three tasks access data in share memory so conflicts may arise when accessing the data concurrently making the data inconsistent.
There are two ways of managing the shared data. In the first case, the resources are actually private for a task and a *resource manager* decides in which order and when the tasks can access the resource. The interaction is done via Inter-Process Communication (IPC). In this case, we don't need mutual exclusion because it is guaranteed by construction since while the manager is deciding which task can access the resource, none can actually use it. Mind that mutual exclusion is solved to access the resources of the tasks, but now the tasks compete to access the resource manager so we need to have mutual exclusion again.
Another possibility is to let the tasks have direct access to a shared memory location. In this case, it is obvious that we need mutual exclusion to access the resource, using, for example, mutexes, semaphores or condition variables which guarantee concurrency.
Since critical sections on the same resource must be executed in mutual exclusion, each data structure should be protected by a mutual exclusion mechanism. This allows to enforce data consistency, but what happens to the real-time performance?
A shared object S_i is said to be a *protected* object when there is a mutex that guarantees mutual exclusion. The mutex associated to the shared resource S_i is usually indicated with S_i too.
We define $cs_{i,j}^k$ to indicate the k-th critical section of τ_i operating on resource S_j. Moreover, the length of the longest critical section of a task τ_i on a resource S_j is denoted as: $\xi_{i,j}$.

```
1  pthread mutex_t m;
2  pthread_mutex_init(&m, NULL);
3
4  void *tau1(void* arg){
5    pthread_mutex_lock(&m);
6    //Execute critical section
7    pthread_mutex_unlock(&m);
8  }
9
10 void *tau2(void* arg){
11   pthread_mutex_lock(&m);
12   //Execute critical section
13   pthread_mutex_unlock(&m);
14 }
```

Let's consider a resource S_1, then the lengths of the two critical sections for `tau1` and `tau2` will be $\xi_{1,1}, \xi_{2,1}$ respectively.
Mutual exclusion on a shared resource can cause **blocking time**, that is when a task τ_i tries to access a resource S already held from another task τ_j, then τ_i needs to wait, i.e., it is moved from the execution state to the block state. Here it will wait for the resource to be unlocked and when it happens it will be moved back into the ready queue, where it will wait to be moved into the execution state again.
The blocking time is the time that passes from the moment in which the task τ_i is blocked to the moment the task τ_j releases the resource. This is needed for implementing mutual exclusion and cannot be avoided. The problem though is that the blocking time might become unpredictable or too large.
Blocking times can be particularly bad in priority scheduling if a high priority task wants to access a resource that is held by a lower priority task. Indeed, in this case a lower priority task would be executing, while a higher one would be blocked. For this reason, schedulability guarantees can be compromised, and we must run schedulability tests taking into account blocking times and give guarantees that they are for sure deterministic and possibly also not too large.
Consider the following example with $p_1 > p_2$:



The white region with an "S" inside is the critical section of the task. It's possible to see that from time 4 to 7, task τ_1 is blocked by the lower priority task τ_2. This is a *priority inversion*, which in this case is not avoidable. The real problem emerges when the priority inversion phenomenon is very large, as in the following case:



with $p_1 > p_2 > p_3$. In such scenario, τ_1 needs to wait from time 4 to 12, having a priority inversion period very large. Notice that what makes such period so long is the fact that τ_2 preempts τ_3 since the former does not need to access the critical section.

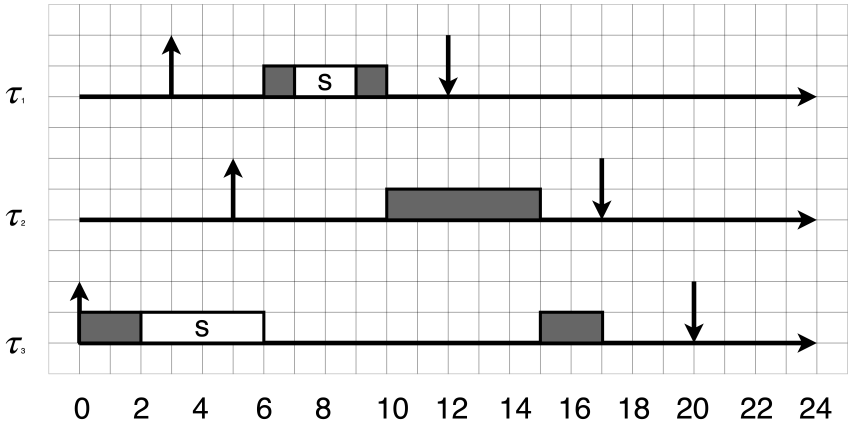
Definition 3.1: Priority Inversion

Priority inversion is a scenario in scheduling in which a high priority task is indirectly superseded by a lower priority task effectively inverting the assigned priorities of the tasks. Inversion occurs when there is a resource contention with a low priority task that is then preempted by a medium priority task.

We want to find a strategy to keep the blocking time small and, in doing so, to reduce the priority inversion. Introducing an appropriate resource sharing protocol allows to find an upper bound for the blocking time.

3.2.1 Non Preemptive Protocol (NPP)

The idea is very simple: when we enter a critical region, we disable preemption. We can do this by raising the priority of our task to the maximum priority in the system when locking the section and restoring the original priority when unlocking. This is very simple to program, but the tasks which are not involved in a critical section suffer blocking.
Let's recall the previous example, using an NPP, we obtain:

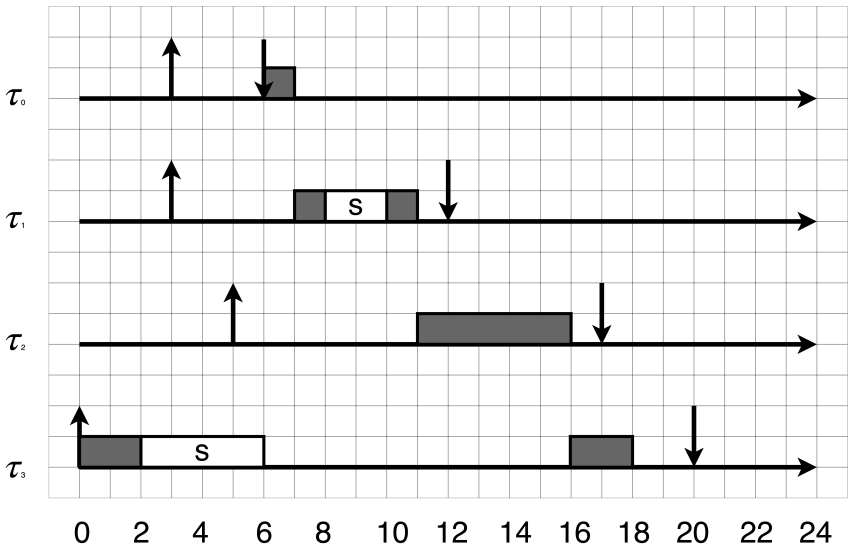


In this way it's easy to see that τ_1 blocking section is greatly reduced, though the problem is moved to τ_2, which has to wait more. This phenomenon on τ_2 is called *indirect blocking* or *push-though blocking*, i.e., it is in the middle between a higher priority task τ_1 and a lower priority task τ_3 both sharing the same resource.

Definition 3.2: Push-through Blocking – Indirect Blocking

A medium-priority job is blocked by a lower-priority job that has inherited a higher priority.

The direct blocking time is bounded by the maximum duration of the critical sections. Also the indirect one should be kept into consideration as shown in the figure below where τ_0 misses its deadline even though its priority is the highest.



3.2.2 Non Preemptive Protocol Improvement

One possible solution to the indirect blocking, is to raise the priority not to the maximum value of the system, but to the maximum value between the tasks which need to access the shared resource. In this way τ_1 and τ_2 would be block, but not τ_0 since $p_0 > p_1$.
The problem is that we need to have a complete mapping of resource usage per each task in order to do a correct scheduling.

So NPPs introduce a blocking time on all the tasks bounded by the maximum length of a critical section used by lower priority tasks. This delay affects the response time in the following way:

$$R_i = C_i + B_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

where

- B_j is the blocking time from the lower priority tasks, and
- $\sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$ maps the interference from higher priority tasks.

3.2.2.1 NPP Example

Let's consider the following tasks table:

Task	C_i	T_i	$\xi_{i,1}$	D_i
τ_1	20	70	0	20
τ_2	20	80	1	45
τ_3	35	200	2	130

Compute the response times of the tasks.
To use the previous formula we need to know also the values of B_i, computed as the maximum value of the critical section times of smaller tasks:

Task	C_i	T_i	$\xi_{i,1}$	D_i	B_i
τ_1	20	70	0	20	2
τ_2	20	80	1	45	2
τ_3	35	200	2	130	0

Then, by using the previous formula we obtain:

- τ_1:

$$R_1 = C_1 + B_1 = 20 + 2 = 22$$

- τ_2, first we consider $R_2 = 0$:

$$R_2^{(1)} = C_2 + B_2 + \sum_{j=1}^{1} \left\lceil \frac{R_2}{T_j} \right\rceil C_j = 20 + 2 + 0 = 22$$

  Then we see if it converges considering $R_2 = 22$:

$$R_2^{(2)} = C_2 + B_2 + \sum_{j=1}^{1} \left\lceil \frac{R_2}{T_j} \right\rceil C_j = 20 + 2 + \left\lceil \frac{22}{70} \right\rceil 20 = 42$$

  Since it has not converged, we set $R_2 = 42$ and we repeat the operation:

$$R_2^{(3)} = C_2 + B_2 + \sum_{j=1}^{1} \left\lceil \frac{R_2}{T_j} \right\rceil C_j = 20 + 2 + \left\lceil \frac{42}{70} \right\rceil 20 = 42$$

  So $R_2 = 42$

- τ_3, first we consider $R_3 = 0$:

$$R_3^{(1)} = C_3 + B_3 + \sum_{j=1}^{2} \left\lceil \frac{R_3}{T_j} \right\rceil C_j = 35 + 0 + 0 + 0 = 35$$

Then we see if it converges considering $R_3 = 35$:

$$R_3^{(2)} = C_3 + B_3 + \sum_{j=1}^{2} \left\lceil \frac{R_3}{T_j} \right\rceil C_j = 35 + 0 + \left\lceil \frac{35}{70} \right\rceil 20 + \left\lceil \frac{35}{80} \right\rceil 20 = 75$$

Since it has not converged, we set $R_2 = 75$ and we repeat the operation:

$$R_3^{(3)} = C_3 + B_3 + \sum_{j=1}^{2} \left\lceil \frac{R_3}{T_j} \right\rceil C_j = 35 + 0 + \left\lceil \frac{75}{70} \right\rceil 20 + \left\lceil \frac{75}{80} \right\rceil 20 = 95$$

We set $R_2 = 95$ and repeat:

$$R_3^{(3)} = C_3 + B_3 + \sum_{j=1}^{2} \left\lceil \frac{R_3}{T_j} \right\rceil C_j = 35 + 0 + \left\lceil \frac{95}{70} \right\rceil 20 + \left\lceil \frac{95}{80} \right\rceil 20 = 115$$

$R_3 = 115$:

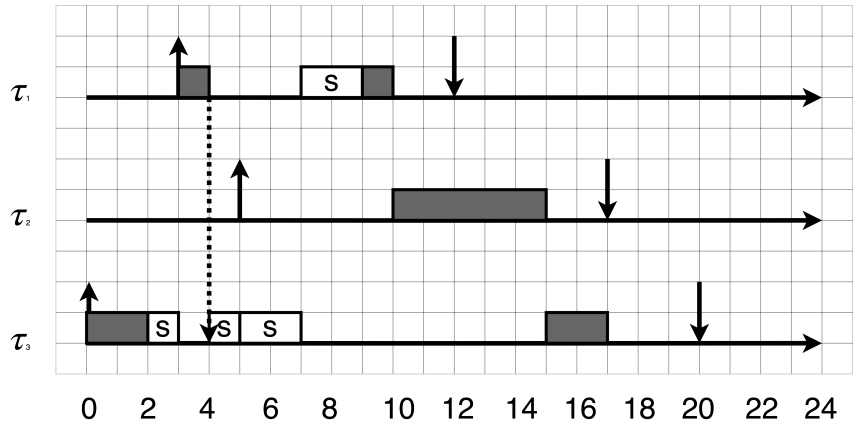
$$R_3^{(3)} = C_3 + B_3 + \sum_{j=1}^{2} \left\lceil \frac{R_3}{T_j} \right\rceil C_j = 35 + 0 + \left\lceil \frac{115}{70} \right\rceil 20 + \left\lceil \frac{115}{80} \right\rceil 20 = 115$$

We have finally reached convergence.

While not strictly a problem, knowing in advance all the resources accessed by the tasks, can be a bit of an hassle. This is a demanding requirement if we want to have a runtime system, since we cannot assume which resource is used by which task.

3.2.3 Priority Inheritance Protocol (PIP)

Another possible solution to the priority inversion is the priority inheritance protocol: whenever a low priority task blocks a higher priority task, it inherits its priority.
Consider the following example:



We can see that when τ_1 starts its execution, it preempts τ_3 since it has a higher priority, but, since τ_3 is already in the critical section, the priority of τ_1 is "copied" to τ_3 (the dashed vertical arrow), so that τ_3 can complete its execution even if τ_2 tries to preempt it, since τ_3 now has a higher priority than τ_2.
Respect to the previous solution, PIP allows for a more dynamic management of the priorities. In a way, the type of information needed is built on the fly as the tasks execute.
The inheritance rules are as follows[1]:

- If a task τ_i blocks on a resource protected by a mutex S, and the resource is locked by a task τ_j, then τ_j inherits the priority of τ_i.
- If τ_j itself blocks on another mutex by a task τ_k, then τ_k inherits the priority of τ_i, called *multiple inheritance*.
- If τ_k is blocked, the chain of blocked tasks is followed until a non-blocked task is found that inherits the priority of τ_i.
- When a task unlocks a mutex, it returns to the priority it had when locking it.

So basically the tasks inherit the higher priority of the tasks competing for a blocked resource.
We would like to show a bounding blocking time also for PIP. To do this, we shall consider only non nested critical sections since in presence of multiple inheritance, the computation of the blocking time becomes very complex.
The maximum blocking time can be computed based on two important properties, which provide an upper bound on the number of times a task can block.

Theorem 3.1

If PIP is used, a task blocks only once on each different critical section.

Theorem 3.2

If PIP is used, a task can be blocked by another lower priority task for at most the duration of one critical section.

The two properties imply that a task can be blocked more than once, but only once per each resource and once per each task.
Then we need to build a resource usage table, where each row is a task and they are ordered by priority, while each column is a resource. Each cell (i, j) contains $\xi_{i,j}$, that is the duration of the critical section for a task i on resource j. Once we have built the table, we can fill it out to compute the bounding time. We know that a task can be blocked only by lower priority tasks, then for each row we must consider only the rows of tasks with lower priority. Moreover, by Theorem 3.1 we know that a task is blocked once, so we can look at the biggest value of lower priority tasks. Finally, we do not have to look at all the columns since a task blocks only on resource directly used, or used by a higher priority task (indirect blocking).

3.2.3.1 PIP Example 1

Consider the following example:

[1]Remember that the indexes of the tasks indicate also the priorities, so the smaller the index, the higher the priority: $\tau_i, \tau_j, \tau_k \Rightarrow p_i > p_j > p_k$

	S_1	S_2	S_3	B
τ_1	2	0	0	?
τ_2	0	1	0	?
τ_3	0	0	2	?
τ_4	3	3	1	?
τ_5	1	2	1	?

We know that a task can either be directly blocked or indirectly blocked, that is, in the former it is blocked because another task is using the resource, while in the latter it is blocked by the fact that a task with higher priority is in the critical section.
Let's now try to compute the B column:

- τ_1 is the task with the highest priority so it cannot suffer from indirect blocking. Moreover, it can be blocked only on S_1, therefore, we can consider only the first column and take the maximum between the lower priority tasks $\max\{0, 0, 3, 1\}$ which is 3, hence $B_1 = 3$.

- τ_2 instead is not the task with the highest priority, so it can be indirectly blocked, hence when computing B_2 we need to take into consideration both the indirect and direct blocking. Let's start by seeing that τ_2 can be blocked only by lower priority tasks, so it can be blocked only by τ_3, τ_4, τ_5. We know that when a task is in a critical section, then it cannot be in another so we should look at the possible combination of two tasks. Moreover, we see that τ_3 is not interested only in S_3 and in neither S_1, nor S_2, hence we can consider only two possibilities:

  - τ_4 accesses S_1 and τ_5 accesses S_2, and
  - τ_4 accesses S_2 and τ_5 accesses S_1.

  In the first case we would have a direct blocking on S_2 of 2, plus an indirect blocking on S_1 of 3, which sums up to $B_2 = 5$. In the second case instead we would have a direct blocking on S_2 of 3, plus an indirect blocking on S_1 of 1, which sums up to $B_2 = 4$. We see then that the first case is the worst, hence $B_2 = 5$

- τ_3: to compute the blocking cost we need to take into consideration both the indirect and direct blocking. Let's start by seeing that τ_3 can be blocked only by τ_4, τ_5, but, differently from τ_2, it can be directly blocked on S_3 and indirectly blocked by tasks accessing S_1, S_2 hence we need to consider: possibilities:

  - τ_4 accesses S_1 and τ_5 accesses $S_2, \Rightarrow 3 + 2 = 5$, and
  - τ_4 accesses S_1 and τ_5 accesses $S_3, \Rightarrow 3 + 1 = 4$, and
  - τ_4 accesses S_2 and τ_5 accesses $S_1, \Rightarrow 3 + 1 = 4$, and
  - τ_4 accesses S_2 and τ_5 accesses $S_3, \Rightarrow 3 + 1 = 4$, and
  - τ_4 accesses S_3 and τ_5 accesses $S_1, \Rightarrow 1 + 1 = 2$, and
  - τ_4 accesses S_3 and τ_5 accesses $S_2, \Rightarrow 1 + 2 = 3$.

  So the value for the blocking on τ_3 is $B_3 = 5$.

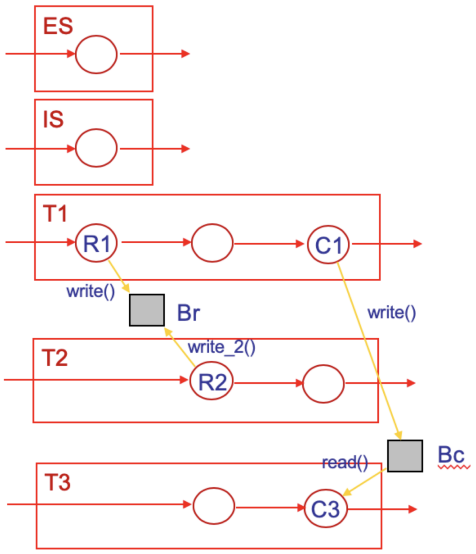
- τ_4 can be blocked on all three columns, therefore we must take into consideration all of them, but can be blocked (directly) only by τ_5. We simply must take the maximum value of the row from τ_5 on all the possible columns: $B_4 = \max\{1, 2, 1\} = 2$.

- τ_5 cannot be blocked by any other task since is the one with the lowest priority, so $B_5 = 0$.

	S_1	S_2	S_3	B
τ_1	2	0	0	3
τ_2	0	1	0	5
τ_3	0	0	2	5
τ_4	3	3	1	2
τ_5	1	2	1	0

3.2.3.2 PIP Example 2

Consider the following example:



We have 5 tasks $\mathcal{T} = \{ES, IS, \tau_1, \tau_2, \tau_3\}$. The last three tasks write and read from two types of shared buffers:

- Result buffers: $R1, R2$ which take respectively $2ms$ and $20ms$.
- Communication buffers: $C1, C3$ which both take $10ms$ to run.

We want to know the blocking times. First of all, we shall create the table:

	C	T	D	$\xi_{1,i}$	$\xi_{2,i}$
ES	5	50	6	0	0
IS	10	100	100	0	0
τ_1	20	100	100	2	10
τ_2	40	150	130	20	0
τ_3	100	350	350	0	10

As we can see, ES and IS do not have any access to the shared resources.
To apply PIP, we can consider only the last two columns and compute the blocking times:

	$\xi_{1,i}$	$\xi_{2,i}$	B_i
ES	0	0	?
IS	0	0	?
τ_1	2	10	?
τ_2	20	0	?
τ_3	0	10	?

ES has the highest priority, but it does not access any shared resource, so it won't be blocked: $B_{ES} = 0$. The same is true for IS: $B_{IS} = 0$.
As for the other tasks:

- τ_1: it uses both resources, so it suffers (direct) blocking on both of them, $B_1 = 20ms + 10ms = 30ms$.
- τ_2: it uses only the first resource. It does not have direct blocking, but it actually has an indirect blocking from τ_3 on the second resource, so the worst case is $B_2 = 10ms$.
- τ_3: having the lowest priority it is not blocked by anything else, so $B_3 = 0ms$.

	$\xi_{1,i}$	$\xi_{2,i}$	B_i
ES	0	0	0
IS	0	0	0
τ_1	2	10	30
τ_2	20	0	10
τ_3	0	10	0

We can then use the computation time to compute the response time analysis:

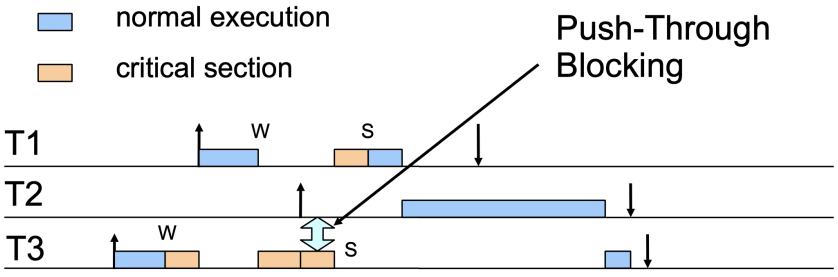
	C	T	D	$\xi_{1,i}$	$\xi_{2,i}$	B_i	R_i
ES	5	50	6	0	0	0	5
IS	10	100	100	0	0	0	15
τ_1	20	100	100	2	10	30	70
τ_2	40	150	130	20	0	10	90
τ_3	100	350	350	0	10	0	300

We could also compute the schedulability test for rate monotonic (RM), indeed the system is schedulable if:

$$\forall i, 1 \leq i \leq n, \sum_{k=1}^{i-1} \frac{C_k}{T_k} + \frac{C_i + B_i}{T_i} \leq i \left(2^{\frac{1}{i}} - 1\right)$$

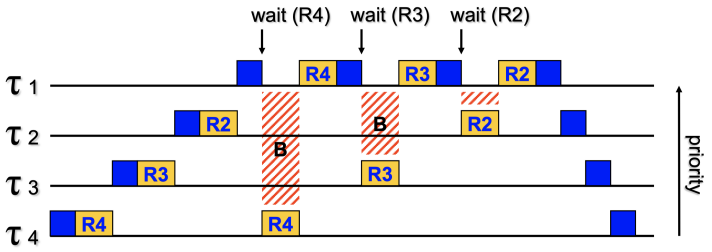
3.2.3.3 PIP Push-through Blocking Example

From the figure below we can see that τ_3 inherits the priority of τ_1, so even if τ_2 tries to preempt, τ_3 can continue its execution. This is noticeable in the so called *push-through blocking*.



3.2.3.4 PIP Example 3

Let's consider the following example:



which is a complicated pattern. Let's look at τ_1, which by the Theorem 3.1 can be blocked at most once per resource and once from any other task. That said, it can still be blocked many times. Indeed, τ_1 would like to access resources $R2, R3, R4$, which instead have all been blocked by other processes priorly. For example, τ_4 starts its execution, enters the critical section to access $R4$, then it gets preempted by τ_3 which has a higher priority and accesses the critical section on another resource R_3. Then τ_2 does the same thing on R_2 and finally τ_1 tries to access $R4$, which is already blocked by τ_4. The last task takes the priority of τ_1, finishes the execution on $R4$, then stops, gets back to its initial priority and τ_1 can access $R4$. The same happens for $(\tau_3, R3), (\tau_2, R2)$. Finally τ_1 executes its last critical section on $R2$, and all the other tasks can terminate their execution following their initial priorities.
While we can manage in terms of computing times, there are some issues. First of all, priorities jump back and forth many times including also frequent context switches, which if were to become too many, would increase the overhead making it no more negligible. Moreover, there are multiple instants of the tasks active at the same time: while if the functions where called one by one, we could share one stack for all the functions, in this case we would need a stack per each function. This is functionally fine, but from the perspective of costs, this could not be feasible.
The disadvantages of PIP are:

- Tasks may block multiple times, resulting in worse performance than with other non preemptive protocols, which maximum blocking time is the maximum duration of the critical section of any task having lower priority.
- Moreover, the implementation is expensive, except for very simple cases.
- Finally, this protocol does not prevent deadlocks.

3.2.4 Priority Ceiling Protocol (PCP)

It comes in two flavours:

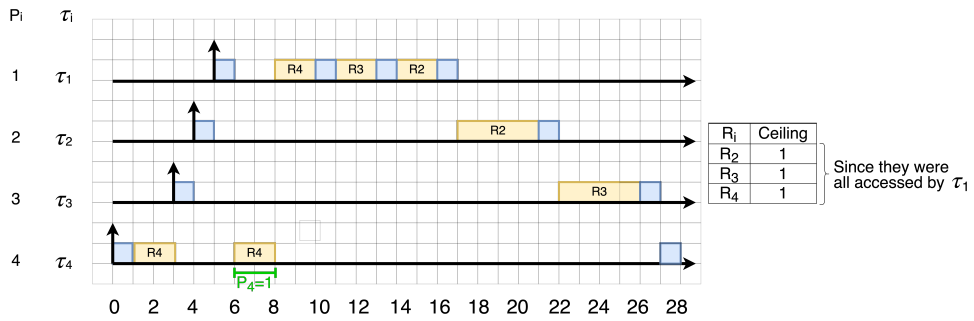
- Original Priority Ceiling Protocol (OPCP)
- Immediate Ceiling Priority Protocol (IPCP)

The idea is to set a *priority ceiling* for a resource, which would be the maximum priority among all the tasks that can possibly access such resource.
Each process is associated with a *dynamic priority* and can only lock a resource if its dynamic priority is higher than the ceiling of any currently locked resource (excluding any that it has already locked itself). So we need to keep track of the resources that are currently occupied so that a task can lock one only if its priority is large enough. If a task τ blocks, the task holding the lock on the blocking resource inherits its priority.
A high priority process can be blocked at most once during its execution by lower priority processes. Moreover, this protocol allows to prevent deadlocks. Also the indirect (transitive)

blocking is prevented minimizing the maximum amount of blocking time, while the mutual exclusion access to resources is ensured by the protocol itself, without using semaphores.

3.2.4.1 Original Priority Ceiling Protocol (OPCP)

Each *resource* has a static ceiling value defined as the maximum priority of the processes that will use it. Each *process* has a static default priority assigned, perhaps using a deadline monotonic scheme. Moreover, a *process* has a *dynamic priority* that is the maximum between its own static priority and any it inherits due to it blocking higher-priority processes. Finally a process can only lock a resource if its dynamic priority is higher than the ceiling of any currently locked resource (excluding any it has already locked itself).
Notice that the priority of a process is not changed to the ceiling priority when locking the resource, but it is changed when the process τ_i blocks another process τ_j, to the priority of τ_j, so the process τ_i can still preempted by other processes.
Consider the previous example. When applying OPCP, we obtain the following:



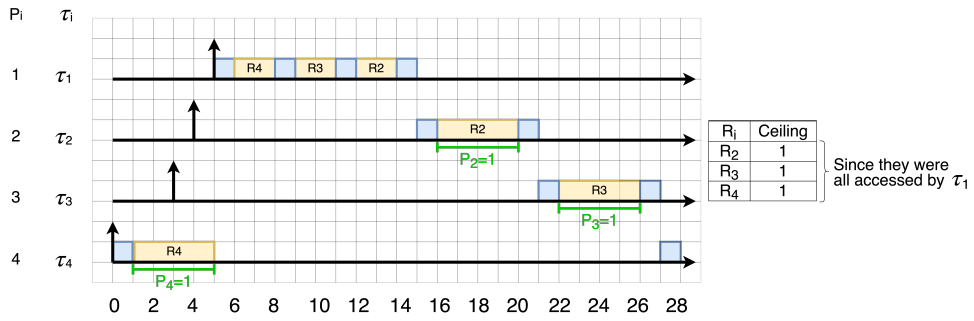
τ_1 needs to access $R2, R3, R4$, τ_2 only $R2$, τ_3 only $R3$ and τ_4 only $R4$. Moreover the tasks are executed in the following order $\tau_4, \tau_3, \tau_2, \tau_1$, while the priorities are $P_1 = 1, P_2 = 2, P_3 = 3, P_4 = 4$. In this case the "highest" priority is the one of τ_1, so all the resources accessed by τ_1 will have its priority (1) as the ceiling, so $R2, R3, R4$'s ceiling priority is 1. When τ_4 starts its execution, there is no locked resource, so it can access $R4$ without problems. Then when τ_3 starts, there is actually a resource locked, so we need to check if its priority is higher than the ceiling of $R4$. It is not, so τ_3 is blocked while τ_4 accesses the resource. While τ_4 blocks τ_3, it takes its priority, so $P_4 = 3$. Then τ_2 starts executing, but when it does, τ_4 releases $R4$, so now there is no locked resource and since τ_3 is preempted by τ_2, it cannot lock $R3$. τ_2 is free to lock $R2$, until τ_1 preempts the process and τ_2 takes the priority of τ_1, which is trying to access $R4$, but the priority of τ_1 is not strictly higher than the one of τ_2, so it is blocked. τ_2 then finishes executing and τ_1 can access the resources and execute. When it finishes, τ_3 is the process with the higher priority and there is no locked resource, so it can execute. Finally τ_4 will end its execution.

3.2.4.2 Immediate Ceiling Priority Protocol (IPCP)

As for OPCP, each *process* has a static default priority assigned, possibly by the deadline monotonic scheme and each *resource* has a static ceiling value defined as the maximum priority of the processes that will use it. A *process* has a dynamic priority that is the maximum of its own static priority *and the ceiling values of any resources it has locked.* This implies two things:

- A process can be preempted by other process with strictly higher priority, not of the same priority;
- A process will suffer a block at the very beginning of its execution.

Once a process τ_i starts actually executing, all the resources it needs will be free for it. Indeed, if they were not, then some process τ_j would have an higher priority and so τ_i wouldn't have started executing.
Mind that a process keeps the ceiling of the resource as its priority for the length of the critical section, then it restore its own priority.
Recall the example from before, if we were to apply IPCP we would have:



The initial setting of priorities and ceilings is done in the same way as OPCP. Then τ_4 starts and since it has the highest priority (better it is the only one that has started), it can access the resource acquiring its ceiling as its own priority, so $P_4 = 1$ for the execution of the critical section of τ_4. While τ_4 is executing, τ_3 would like to start but it cannot since τ_4 has a higher priority. When τ_4 ends the critical section, it restores its own priority $P_4 = 4$, but at this point, τ_1 starts its execution and has a higher priority than both τ_2 and τ_3 hence it will execute blocking first R_4, then R_3 and finally R_2. Once τ_1 is done, τ_2 starts and move its priority to 1 when entering the critical section (even though it was not necessary). Then τ_3 executes and finally τ_4 can end its execution.
Although the worst-case behaviour for OPCP and IPCP is identical from a scheduling view point, there are some differences:

- IPCP is easier to implement than the original version as blocking relationships don't need to be monitored;
- IPCP leads to fewer context switches as blocking is connected to beginning the execution;
- IPCP requires more priority movements as this happens with all the resources, whereas OPCP changes priority only if an actual block as occurred.

3.2.4.3 Example

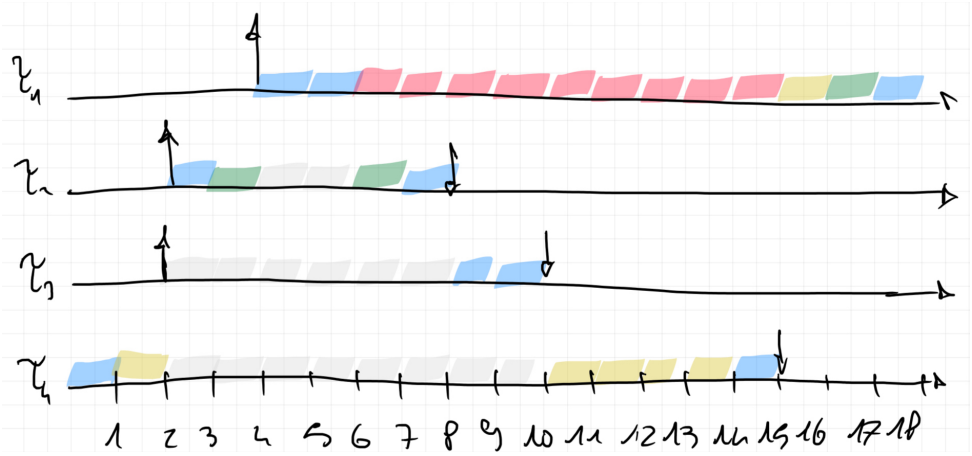
Consider the following table for the processes representing their priority, their release time and their execution sequence:

τ_i	P_i	R_i	Exec Sequence
τ_1	4	4	EE Q V E
τ_2	3	2	E VV E
τ_3	2	2	EE
τ_4	1	0	E QQQQ E

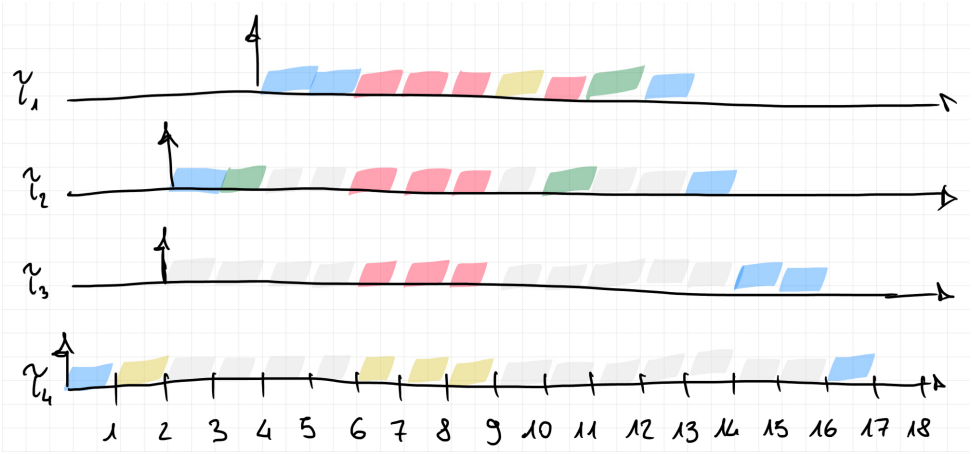
Moreover the following colors represent:

- Preemption;
- Blocking.

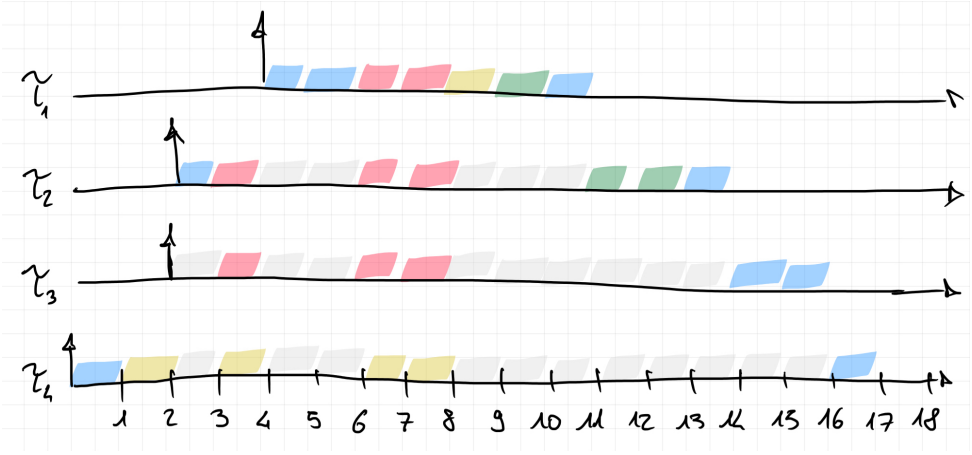
When using only resource sharing, then any process gets blocked by another process accessing the shared resource:



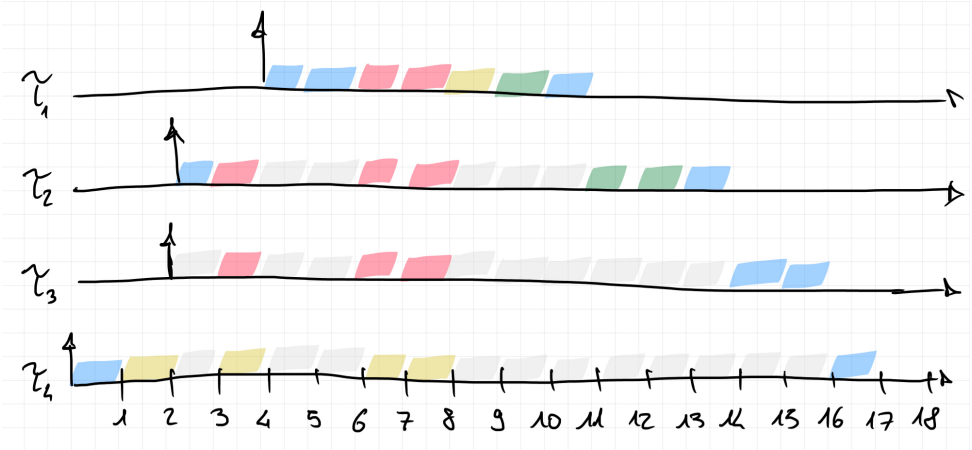
Instead when using priority inheritance, when a process τ_j by another process τ_i already accessing the resource, τ_i acquires the priority of τ_j, hence preempting other processes that may have been running previously of τ_1 beginning.



With OPCP, the processes only inherit a priority when blocking another process and only for the critical section period, moreover they can access a shared resource only if their dynamic priority is higher than the ceiling of any priority which is already being used. In this case there are two resources, Q and V. They are both accessed by τ_1, so their ceiling will be 4 for both.



With IPCP, the processes only inherit a priority when accessing a shared resource and so only for the critical section period. Since they get the ceiling of the resource as their priority, the protocol ensures that they are the only process that could get the access in that moment.



4 Dynamic Priorities

We shall now go back to processes which do not interact among each other.
We have seen that rate monotonic (RM) and deadline monotonic (DM) are optimal fixed priority assignments. So if there exists a possible schedule for a fixed priority assignment, then there is one with one of the two.
But what happens when the priorities are dynamic? With fixed priority scheduling, a task τ always has the same priority during the life of the system, but with dynamic priority scheduling, the priority of a task τ may change during the lifetime of the task. We shall assume that priorities change from job to job[1], while a precise job $J_{i,j}$ always has the same priority $p_{h,k}$. So, given a task τ_i, its priority p_i can change, whereas the priority $p_{i,j}$ of a single job $J_{i,j}$ is constant.

4.1 Earliest Deadline First (EDF)

The simplest idea is to give priority to tasks with the earliest absolute deadline (which is given by the start time plus the relative deadline):

$$d_{i,j} < d_{h,k} \Rightarrow p_{i,j} > p_{h,k}$$

This approach is called **Earliest Deadline First (EDF)** and we must use absolute deadlines and not relative ones since the former changes each time the task is activated, while the latter does not. If we consider a system of periodic tasks with relative deadline equal to the period, then the system is schedulable with EDF if and only if:

$$U = \sum_i \frac{C_i}{T_i} \leq 1$$

When we work on a single processor with the given assumptions, the optimal is obtained when the previous sum is 1 and EDF does this. EDF is "better" than rate monotonic since having a system with dynamic priorities allows for more freedom and hence a better system.
If the deadline is different from the period, then we need to resolve to time demand analysis or to response time analysis, but it can be complex.

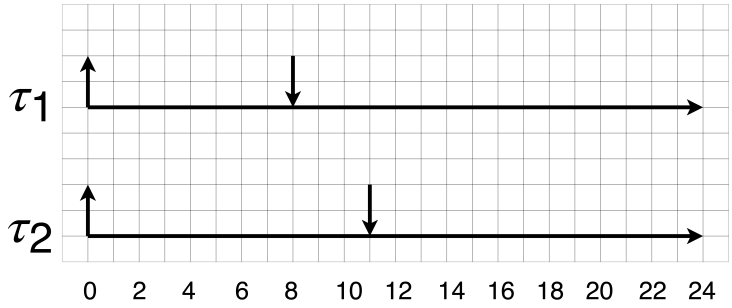
4.1.1 EDF Example

Consider the following task set $\mathcal{T} = \{\tau_1 = (3, 8, 8), \tau_2 = (6, 11, 11)\}$. By the previous formula, we get that the utilization is:

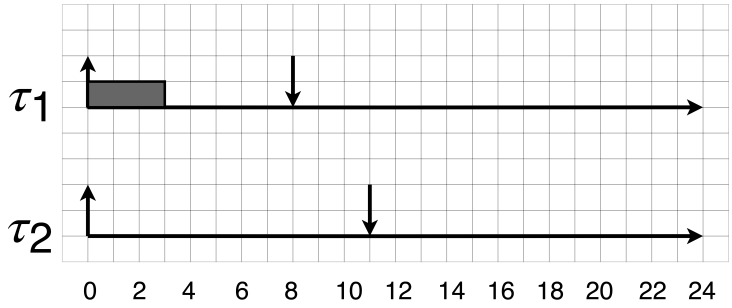
$$U = \sum_i \frac{C_i}{T_i} = \frac{3}{8} + \frac{6}{11} = 0.92$$

At the beginning, both rate monotonic and EDF are the same:

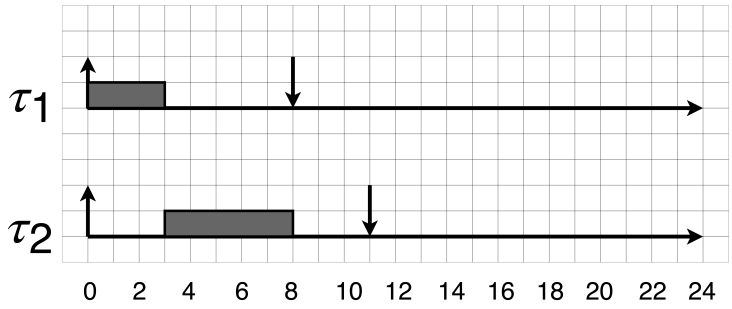
[1] Tasks are made of jobs.



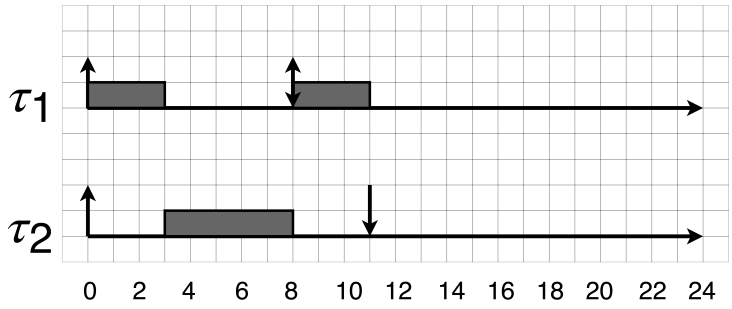
The first one to be scheduled is τ_1:



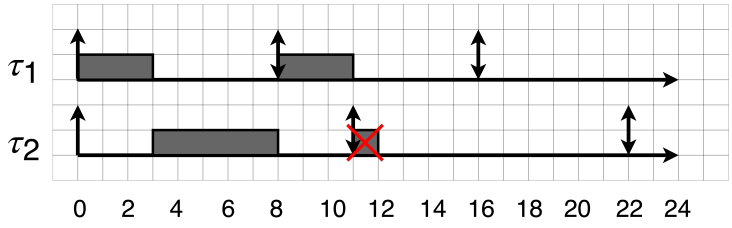
Then τ_2 can execute:



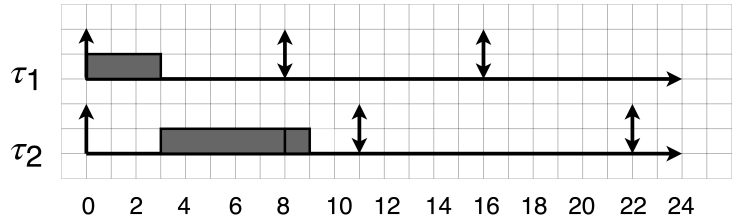
But while the second task is executing, it gets preempted by τ_1. If our scheduling protocol is RM, then τ_1 takes control and executes, while τ_2 waits:



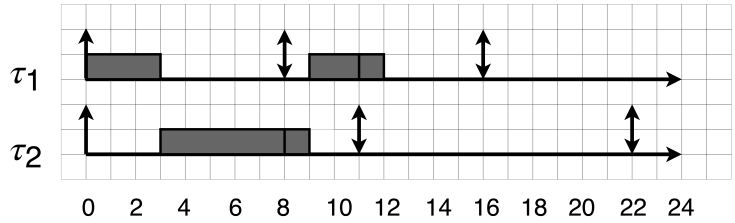
Then the second task can resume and end its computation but it cannot make the deadline:



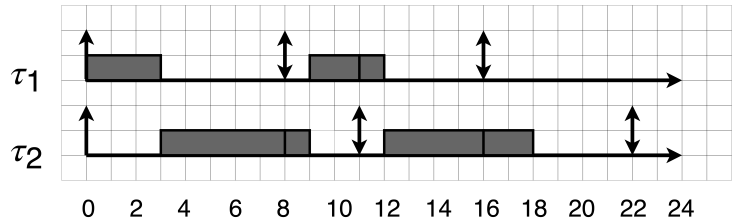
Instead if we were using EDF, we would have checked which task would have ended first (in absolute terms), which is τ_2, so it would have been allowed to continue its execution:



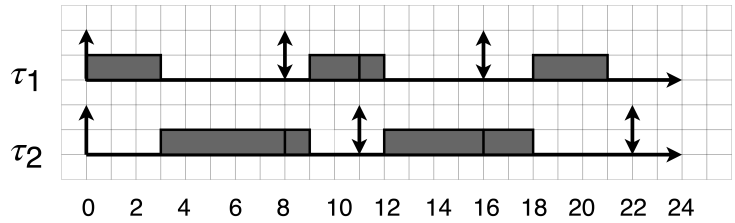
Then τ_1 can start its execution, but gets interrupted by τ_2. We shall check which will end first, and it's obviously τ_1, so it will continue executing:



Then τ_2 can execute, but again, it will be interrupted by τ_1, which absolute deadline is later in time than τ_2's hence, τ_1 can continue executing:



Finally τ_1 will execute:



From a schedulability point of view, EDF is wonderful, but not many operating systems have tried to support this scheduling policy and only recently Linux, with the 3.14 version has started its support.
The way we map priorities is easier to be done with fix priorities than with EDF, and this is one of the point that always kept EDF behind. Moreover, the cost of computing the correct priorities each time a new task starts executing should be kept into consideration (especially in smaller systems), when instead fixed priority is constant in time.
The advantages for EDF is its schedulability while the disadvantages are the computational complexity in the implementation of the queues and the difficulties in implementing time demand analysis and response time analysis.

5 Real-Time in the Real World

We have created a model, we shall now discuss if the assumptions made are actually realistic. Indeed we have talked about a set of independent tasks $\mathcal{T} = \{\tau_i, i \in \{0, \dots, N\}\}$. We have said that a task is periodic, often assumed that deadline and period correspond and we used only one CPU. One problem we have not managed to tackle yet is the Worst-Case Execution Time (WCET): we have seen that it actually depends by many factors, from how the tasks work, to the system condition (memory, stacks, etc).
In practice, we have a real-time system where the tasks can be also sporadic. This introduces another problem that is the Minimum Inter-arrival Time (MIT). Moreover, in real-time systems, we should consider scheduling on more than one CPU.
Finally we should also take into consideration the operating system overhead.

5.1 Worst-Case Execution Time (WCET)

The schedulability analysis is based on the WCET. We can estimate its value in two ways:

- **Sensitivity Analysis**: profile the execution time of the task multiple times with different inputs trying to cover all the possible inputs. By considering all the possible profiled times, we can end up with a too big time wasting resources or we can end up with a too small time.
- **Resource Reservation**: limit the execution time in some way, enforcing a WCET. Since we cannot be sure that the execution time stays the same, we can put cap the maximum execution time of the task.

5.1.1 Sensitivity Analysis

If a job $J_{i,j}$ executes for a time $c_{i,j} > C_i$, we have to formulate the problem as a TDA, or RTA approach and see if the error can actually disrupt schedulability.
This is not easy to do since the response time of a task is a function of the computation time and the periods, hence not linear:

$$R_i = f(C_1, \dots, C_i, T_1, \dots, T_{i-1})$$

Moreover, the function can be studied with the bifurcation theory, which is the study of changes in the qualitative or topological structure of a given family, which leads to a complex analysis of the function.

5.1.2 Reservation-Based Scheduling

This is probably the most used in many cases. The idea is to force the task to not demand more time than the system has reserved to it for a given period. That is, the system reserves enough time as if it was reserving the space for a periodic (or sporadic) task $\tau_d = (Q, T)$, and forces a

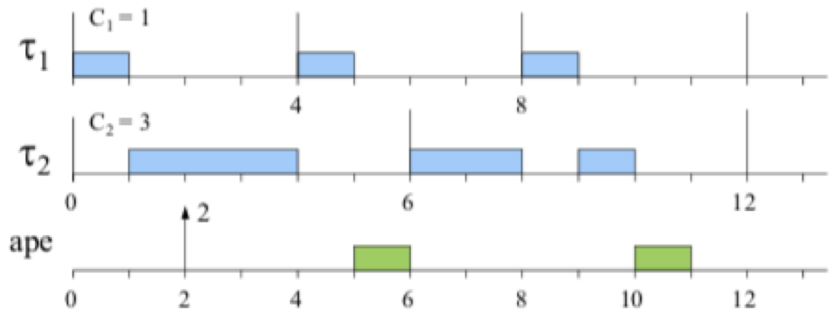
task τ_i to remain in those temporal constraints for that given period. If the task oversteps, the system is not obliged to give any more resources to the task and its execution may as well be postponed to the next period. This is similar to *traffic shaping* in networks.
From the perspective of the other tasks, the system is protecting them from the misbehaving tasks, that is tasks that executes for too much time or if the computation of the WCET is wrong. A misbehaving task might miss the deadline, but the other are guaranteed to start on time. This is called **temporal protection**.
Let's say that the system has a budget q which is consumed when the task executes. When the budget is 0, the task cannot be scheduled anymore. The task can be *replenished* when the task ends and the system ask for the resources back. The system also has an *accounting* mechanism.

5.2 Aperiodic Tasks

When we don't know anything about the interarrival times, we cannot give guarantees. So how do we deal with aperiodic tasks.
One solution is to accept the fact that they cannot have guarantees, so we simply schedule them in the background in the sense that we first concentrate on the periodic ones and if there are resources left, then we can schedule the remaining tasks. This though implies that the aperiodic tasks may be delayed a lot, while some aperiodic tasks are still urgent to be executed, for example the input from a keyboard.
The traditional solution is to use a periodic task (or sporadic) task to serve the aperiodic requests. In this case the periodic task is called **aperiodic server**, of which there are various, for example: polling server, deferrable server, sporadic server, etc.
Aperiodic tasks are activated by events which do not take place on a regular basis. There are mainly two situations:

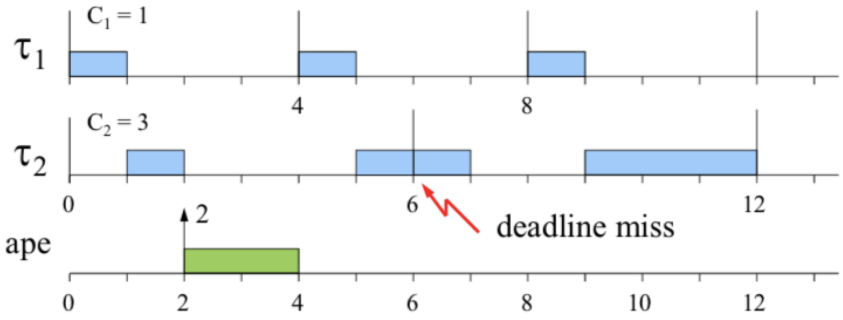
- We have temporal guarantees on the aperiodic tasks, i.e., there is at least a certain amount of time between two subsequent events. If this was the case, then we could assume aperiodic task similar to sporadic tasks and fallback to the previous theory;
- We do not have them.

Notice that for sporadic tasks we need to know the WCET and the MIT: with this knowledge we can actually formulate guarantees, while in its absence we can only try our best.
We want to treat aperiodic tasks "nicely", in the sense that we want to minimize their response time without violating the guarantees of periodic hard tasks.
We have said that one approach is to schedule the aperiodic task in the background and when the CPU is left idle, we try to fit in the execution of the aperiodic task.



In this case, if we assume that the computational time of the aperiodic task is 2, then its response time is 9 since it arrived at $t = 2$ and was completed at $t = 11$.

Another possibility would be to execute the aperiodic task with the highest priority, that is, when it pops up, it preempts the processor and is immediately run:



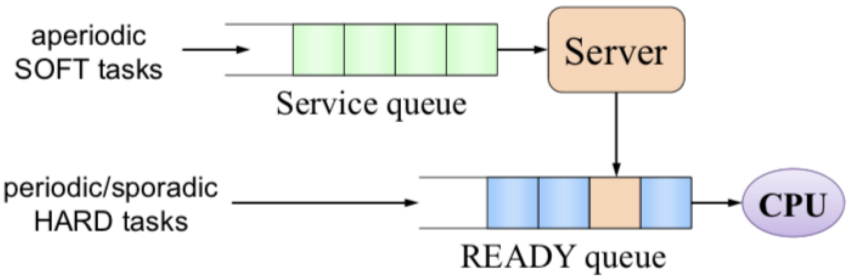
While in this case the response time of the aperiodic task is minimized, τ_2 ends up missing a deadline.

5.2.1 Aperiodic Server

One possible solution is to use an aperiodic server, that is an algorithm in which we try to model the execution of an aperiodic task as embedding it in the execution of a periodic task.
Normally we associate two parameters with a server:

- C_s: the capacity of the server;
- T_s: the period of the server.

The idea is that the served tasks receive no more than C_s time units every T_s.
Generally speaking, the structure of an aperiodic server is the following:



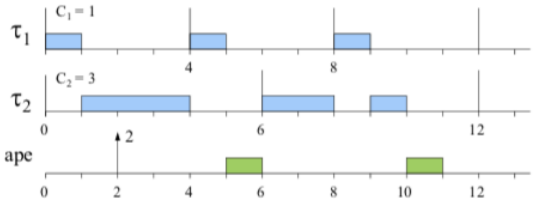
The aperiodic tasks are stored in a service queue from where the server picks them up one by one and it searches for the right point in the ready queue where to insert the aperiodic task. This means to select an appropriate priority for the server and also that we need to manipulate the priorities to accommodate the server. Then the problem becomes the order of the aperiodic tasks in the queue: typically any arbitrary discipline is fine, usually FIFO is used.

5.2.1.1 Polling Server

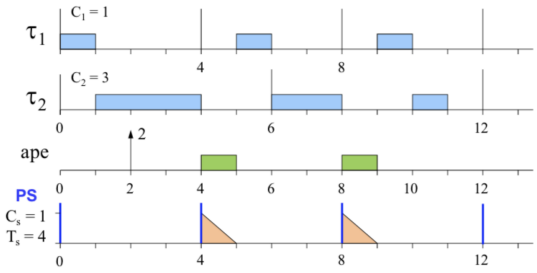
The server has a capacity C_i and a period T_i: each time the server is activated at a new period, it checks for pending tasks:

- if there is no pending aperiodic task, then C_s is set to zero; otherwise
- if there are pending jobs, they are allowed to execute depleting C_s until it becomes 0.

Every time the server period expires, the capacity is recharged to its value and it decreases while the server executes.
Let's recall the example from before shown on the left below. If we were to use a polling server with rate monotonic, then we would have the graph shown on the right. This allows to have a better response time for the aperiodic task, while keeping the periodic tasks within the deadlines. It's possible to see that the polling server has a period of 4, so every 4 units of time, the server capacity is replenished. Moreover, we can see that at time 0, there is no waiting aperiodic task and hence the capacity is simply thrown away, while on time 4, there is an aperiodic task waiting. Since our capacity is 1, then only one part of the aperiodic task is executed, while the rest must be scheduled for the next period of the polling server. This leads to a response time of 7 because the aperiodic tasks started at time $t = 2$ and ended at time 9.



Response time: 9



Response time: 7

In the worst-case execution time the server behaves as a periodic task with utilization:

$$U_s = \frac{C_s}{T_s} \tag{5.1}$$

Aperiodic tasks execute at the highest priority if T_s is the minimum period.
To see the impact on the other tasks, it is possible to compute U_{lub} taking into consideration **??**:

$$U_{lub}^{RM+PS} = U_s + n\left[\left(\frac{2}{U_s+1}\right)^{\frac{1}{n}} - 1\right] \tag{5.2}$$

Then we need to analyze how the server behaves with respect to its aperiodic task. We have to do an analysis of what its maximum computation time would be: suppose that the task is activated at a certain point after the first period of the server, then there is an *initial delay* Δ_a which must be accounted for since the server is not able to respond as it threw the capacity away and it's waiting for the next period.

$$\Delta_a = \left\lceil \frac{r_a}{T_s} \right\rceil T_s - r_a \tag{5.3}$$

where r_a is the initial response time of the aperiodic task (which is missed) and T_s is the period of the server.
Then we need to compute the *full service periods* F_a, that is how many server periods are needed since each time the server is recharged, we can serve a small amount of time:

$$F_a = \left\lceil \frac{C_a}{C_s} \right\rceil - 1 \tag{5.4}$$

where C_a is the computation time of the aperiodic task and C_s is the server capacity.
Finally we need to compute the *final chunk* δ_a, that is the remaining piece of computational time of the aperiodic task to be executed:

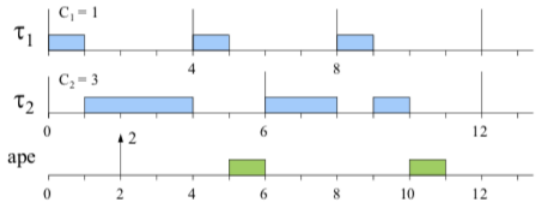
$$\delta_a = C_a - F_a C_s \tag{5.5}$$

Finally considering (5.3, 5.4 5.5), we can compute the *response time* of the aperiodic task as:

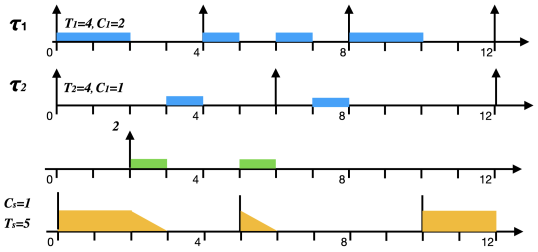
$$\begin{aligned} R_a &= \Delta_a + T_s F_a + \delta_a \\ &= \Delta_a + T_s F_a + C_a - F_a C_s \\ &= \Delta_a + C_a + F_a(T_s - C_s) \end{aligned} \tag{5.6}$$

5.2.1.2 Deferrable Server

The fact that we empty the capacity when the server is idle, introduces a large delay. We could avoid the budget being depleted and then let the system go as with polling server. Only when the capacity is depleted, the server stops and waits for the next period to recharge its capacity. We shall confront the previous example (left) with the deferrable server solution (left):

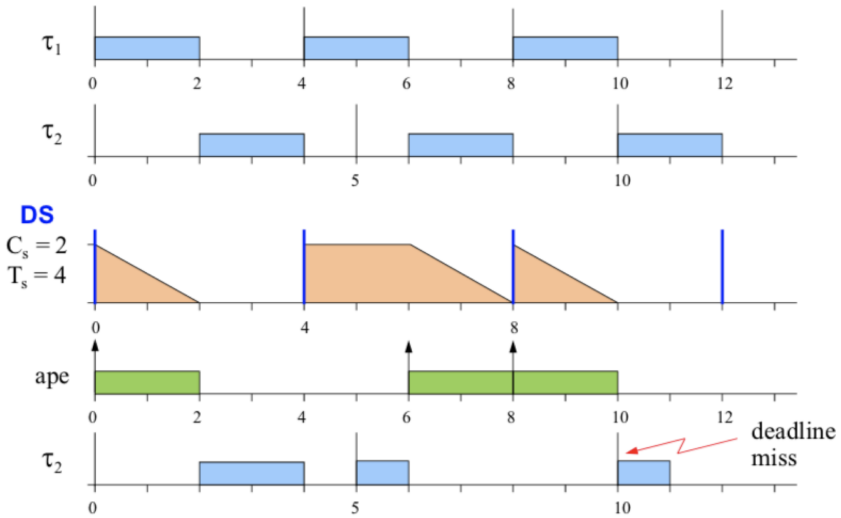


Response time: 9



Response time: 5

We can see that, even if there is no aperiodic task ready to start at the first period of the server, the capacity is not thrown away, but instead it is kept and, when the aperiodic task pops up in the middle of the period, it is run until it depletes. In this way, the response time of the aperiodic task is 5.
The idea is nice, but the problem is that the deferrable server does not always work as a periodic task, hence there may be problems. Indeed, considering the below example, if we were to change τ_1 with an aperiodic task and a deferrable server with the same parameters of τ_1, we would see that τ_2 will miss a deadline, showing that a deferrable server is not always the same as a periodic task.

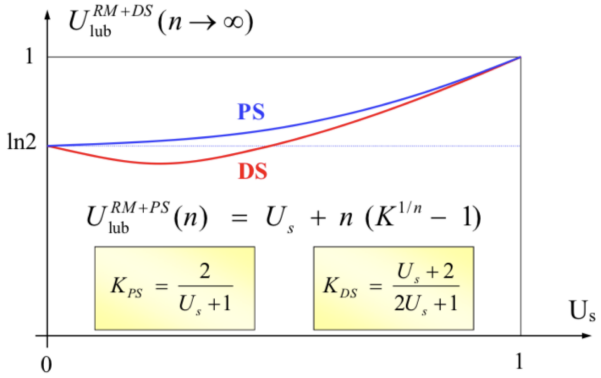


When the server starts, the aperiodic task starts too, so the capacity of the server is immediately used. Then at time 4, the capacity is recharged to 2 and its held constant since there is nobody that wants to execute. When the aperiodic task happens, it preempts task τ_2. Then at time 8 the server is recharged again and immediately the aperiodic task happens leading to a deadline

miss for τ_2.
We can compute the least upper bound of the deferrable server as:

$$U_{lub}^{RM+DS} = U_s + n\left[\left(\frac{U_s+2}{2U_s+1}\right)^{\frac{1}{n}} - 1\right] \tag{5.7}$$

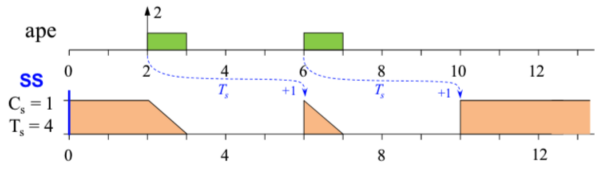
By comparing the function of the least upper bound of the deferrable server with the one of the polling server, we have:



The utilization remaining for the deferrable server is below the ones of the polling server, which means that the polling server leaves more room for the execution of the periodic tasks.
On the other hand, if we compute the response time, we find out that it is shorter than the polling server response time since we do not waste time to wait for the next recharge of the capacity.

5.2.1.3 Sporadic Server

Like the deferrable sever, the unused capacity is not thrown away, but we delay the recharge time. So it preserves the budget and this is recharged after the period T_s only if the capacity is finished.
The sporadic server is not activated periodically, but it behaves as a periodic task with computation time C_s and period T_s.



So the recharge time is set to be T_s time units after the activation of aperiodic task.
Let's define:

- P_{exe} the priority level of the executing task;
- P_S the priority level associated with the sporadic server;
- `ACTIVE`: the sporadic server is said active if $P_{exe} \geq P_S$. The server is flagged as active both when it is executing and when some other task with higher priority is;
- `IDLE`: the sporadic server is said to be idle if $P_{exe} < P_S$. Notice that this implies that the server might still be active even if the capacity is depleted, but there are some other tasks still executing with an higher priority;

- RT the replenishment time, i.e., the time at which the capacity is replenished;
- RA the replenishment amount, i.e., the amount of the capacity that is replenished;
- T_S, C_S the period and the capacity of the server.

We define also two rules:

Definition 5.1: Rule 1

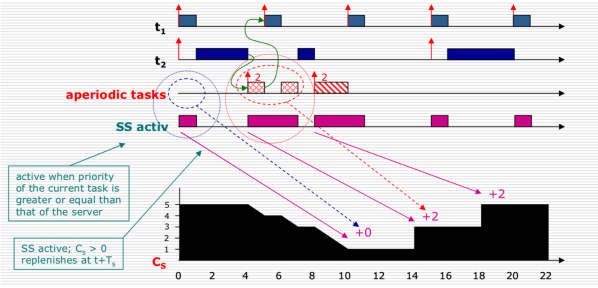
Let t_a be the time in which the sporadic server becomes active, then we set the replenishment time $RT = t_a + T_S$;

Definition 5.2: Rule 2

Let t_I be the time when the server becomes idle or the capacity is exhausted, then we set RA equal to the amount of time consumed in $[t_a, t_I]$.

5.2.1.3.1 Example 1

Let's consider two tasks $\tau_1 = (1, 5), \tau_2 = (4, 15)$ and a sporadic server $\tau_S = (5, 10)$. Considering a rate monotonic priority, then the order of the tasks is: τ_1, τ_S, τ_2.



When τ_1 starts executing, its priority is higher than the one of τ_S, so for the first rule, the server becomes active and we schedule a replenishment point to the current position plus the period of the server:

$$RT = 0 + 10 = 10$$

Then τ_1 completes and leave space for τ_2 that has a lower priority than the server. Moreover, the server does not have any pending activity to be done, so, by rule 2, it will transit from an active state to the idle one. We can now compute how much server time we have consumed in this interval of time and get the amount of replenishment. Notice though that the server while active did not execute any aperiodic task, but only τ_1 executed, so the replenishment amount is 0. Indeed it's possible to see that at the first period, there is no replenishment.
It's possible to see that at time 4, an aperiodic task arrives and since the sporadic server has a higher priority than τ_2, it preempts the process, moves into an active state and executes the aperiodic task. Since the server has become active, we are going to schedule a replenishment point:

$$RT = 4 + 10 = 14$$

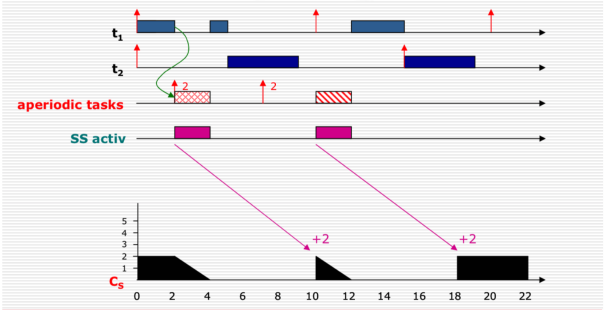
Then at time 5, τ_1, which has a higher priority, preempts the sporadic server and executes. The server remains active due to the first rule and then continues the execution of the aperiodic task. It's possible to notice that the active time is 3, but the period of time in which the server actually executed is 2, so the replenishment amount at time 14 will be 2.

5.2.1.3.2 Example 2

In this case the task set is the following:

$$\mathcal{T} = \{\tau_S = (2, 8), \tau_1 = (3, 10), \tau_2 = (4, 15)\}$$

So the server has the highest priority.



As before, τ_1 starts, but in this case the priority of τ_1 is lower than the one of the server, so the server does not move to an active state but it stays idle.
At time 2 the server gets activated, so we have a transition from idle to active and we schedule a replenishment time at time

$$RT = 2 + 8 = 10$$

Moreover we can see that we have consumed 2 units of time, so the replenishment amount will be 2. Notice that even if the aperiodic task was longer than 2, the sporadic server capacity is 2, so the task would have been interrupted until the server was replenished.

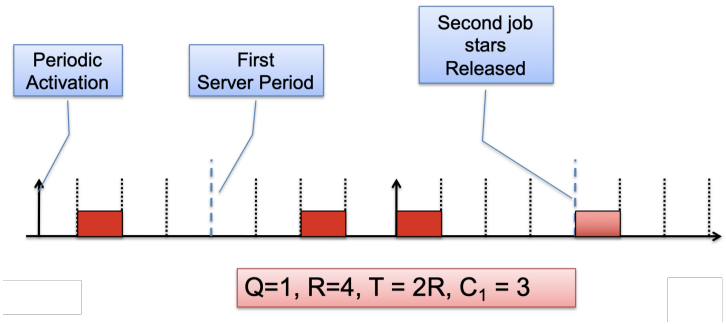
5.2.2 Overruns – Temporal Isolation

The sporadic server features are very good since it strikes a good trade-off between treating well the aperiodic task with a relatively short response time and in the meantime trying to preserve the execution time of the other tasks. Moreover it also avoids the problem of back to back execution.
Let's recall a problem we have talked at the beginning of the chapter: what should happen if a job executes more than expected, so if it depletes the entire capacity? The desired behaviour would be to suspend the task that is overrunning the server and at the same time we would like all the other tasks to continue their execution. So we would like to have some temporal fire-walling to guarantee that the task does not receive more than it should and, if it does, we would like it to be the only one that pays for its own delays.
Until now we have seen how to give some guarantees to the aperiodic tasks, but now we want to make a good compromise between the periodic and aperiodic tasks. This implies handling situations that frequently happen in reality. In these cases, we usually know the computation time and if it is periodic or not, so we can apply the classical real time theory, but we'll see what happens if we are making a mistake.
We have seen that we could do time demand analysis and estimate how good our model is, but if an overrun happens, then we go behind the schedulability region since all the tasks will be messed up. In this case, we know that a task may misbehave and we are encapsulating the task in a layer that guarantees that if the task misbehaves, it will not destroy the normal flow of execution.
The way to achieve this is by doing **temporal isolation**. We are going to use a server again, but not to guarantee quick response time to the aperiodic task without considering the needs of periodic task, but we are using a server to enforce temporal constraints. The algorithm has an

enforcement mechanism (on top of the scheduler) such that we have a cut-off if we exceed the bandwidth, i.e. a fraction (percentage) of the execution of the processor. Each task is given a percentage of the processor it can use and if it exceeds it, then the enforcement mechanism gets in action.

5.2.2.1 The Resource Reservation Paradigm

By the resource reservation paradigm, we ensure that each task receives Q_s time units in every reservation period T_s.



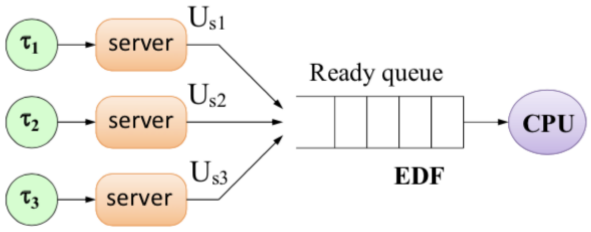
where R is the reservation period and T the task period. These don't have to be the same, indeed usually the reservation period is a sub-multiple of the task period.
It's possible to see that the task is not scheduled in the same position in each period, but the guarantee is that it is going to be scheduled and that it won't overrun.
Changing R allows us to change the allocation of the task:

- If it goes to 0, then the allocation mechanism is more fluid;
- On the other hand, if it becomes too small, the overhead introduced by the scheduling mechanism becomes too high.

We shall choose the reservation period has a fraction of the period of the task.

5.2.2.2 Constant Bandwidth Server (CBS)

This is an idea on how to implement the resource reservation paradigm. It is based on an Earliest Deadline First (EDF) scheduler, where each task is encapsulated within a server that enforces temporal protection. A bandwidth is assigned to each temporal server and the EDF server is in charge to select the order in which the tasks are going into the ready queue and then passed to the CPU.



Notice that there also could be different tasks making requests to the same server.
The CBS is a way to obtain the resource reservation mechanism by using dynamic priorities and manipulating them to enforce the temporal constraints.

The job executions are monitored through a budget mechanism and when the budget is exhausted the deadline is postponed. So this server works by manipulating both the priorities with the EDF and also by maintaining a budget mechanism.
The CBS is associated with some parameters that can be divided into:

- Assigned by the user to each task and are fixed:
  - Q_s: maximum budget, that is the capacity of the server;
  - T_s: server period;
  - $U_s = \frac{Q_s}{T_s}$: bandwidth;
- Managed by the server:
  - q_s: current budget;
  - d_s: scheduling deadline.

Both q_s and d_s are initialised to 0. The goal is to reserve Q_s units of time for the execution each T_s units of time.
The variables managed by the server are state variable and are used to implement the CBS mechanism.
The rules of the server are the following:

Definition 5.3: Rule 1

At the arrival of a job J_k at time r_k, we need to assign a deadline d_s. We do so by checking if there is any pending execution request: if there is, then the server becomes active and we shall enqueue J_k. If not, then we shall check if the currently used budget is enough to let the task run, if it is then we continue to use the same budget with the same deadline, otherwise we assign d_s to the time from now plus the server period:

$$d_s = r_k + T_s$$

and we also update the current budget to the maximum capacity.

if $\exists$ *pending aperiodic request* **then**
  enqueue(J_k)
else
  if $q_s < d_s - r_k)U_s$ **then**
    $q_s = Q_s$
    $d_s = r_k + T_s$
  end
  continue to use q_s, d_s
end

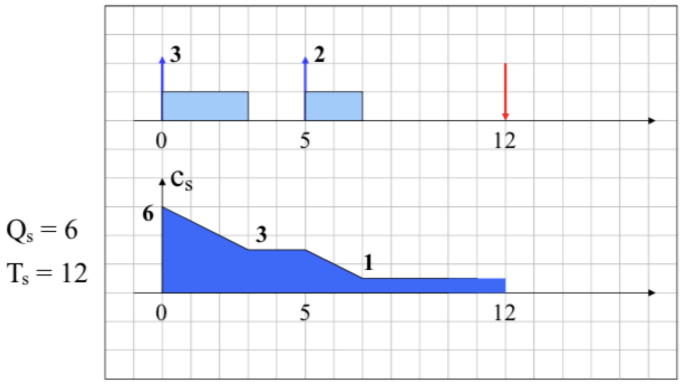
Definition 5.4: Rule 2

When the budget gets depleted, we need to recharge it and postpone the server scheduling deadline:

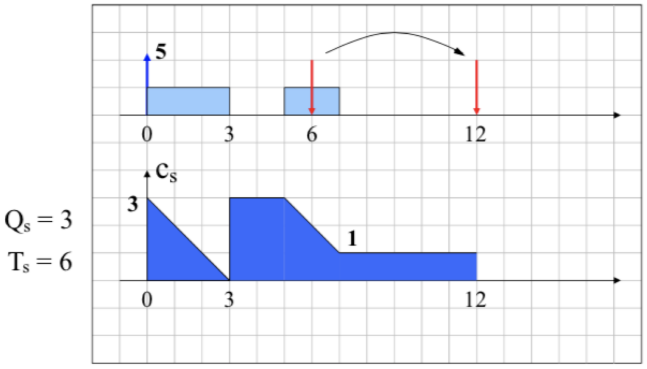
$$\begin{cases} q_s = Q_s \\ d_s = d_s + T_s \end{cases}$$

The two rules work on:

- activation: we try to see if we can use the old capacity and deadline and if not we recharge it;

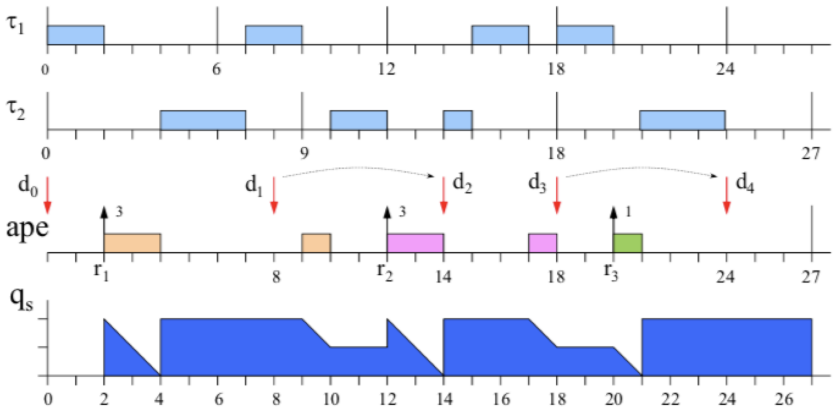


- budget exhaustion: recharge the capacity and postpone the deadline.



There is a variant to CBS, which states that once the capacity is depleted, it won't get recharged until the deadline expires. This variant is called *hard reservation* and is the one implemented in Linux. In this case we guarantee only Q_s units of time for each period, so we fix $^{Q_S}/_{T_S}$, while on the soft reservation we guarantee at least Q_s units of time. Moreover, notice that if a task has parameters (C, T, T) with utilization $^{C}/_{T} = U$ and is scheduled with an hard CBS with parameters (Q_s, T_s) with $^{Q_s}/_{T_s} > U$ and T_s is equal to T or a submultiple, then if the total assigned bandwidth is below 1 and the task will meet all of its deadlined no matter what the other tasks in the system do.

5.2.2.2.1 Example of EDF and CBS

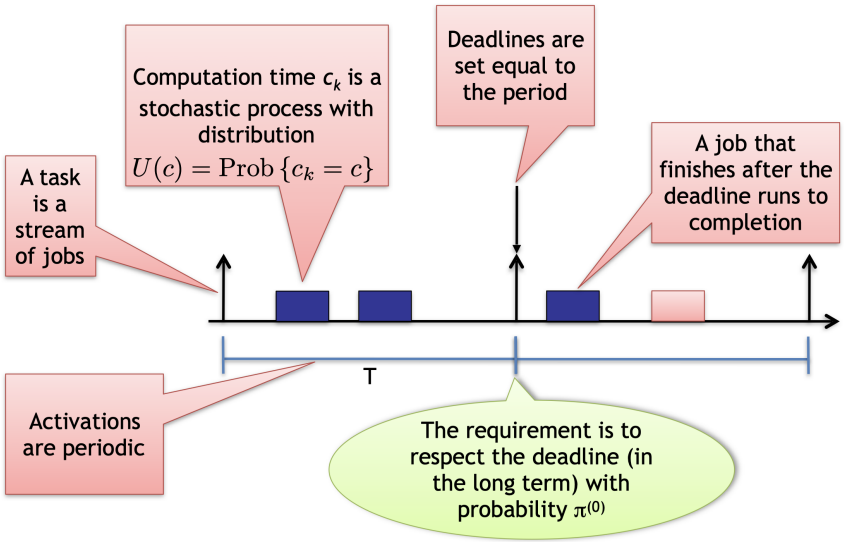


CBS: $Q_s = 2$, $T_s = 6$

We can see that there are two tasks that are scheduled with EDF and then there is an aperiodic task which instead is scheduled with CBS.

5.2.2.3 Probabilistic Guarantees

Consider a soft real time task for which the strict respect of all deadline is not required as long as the task respect some kind of "good" probabilistic behaviour. On one end we do not care to have 100% guarantees, but at the same time we still need to give guarantees, hence we could do so by describing a probabilistic behaviour.
In this way we have the possibility to associate each task with a probability of respecting the deadline. There are some tasks, even important that can still miss some deadlines without problems.
In this case we don't have the knowledge of the WCET, but we know the probability distribution of the execution time of a task. The model that we should define is the following:



There are some observations to make:

- Temporal isolation property allows us to model the evolution of the tasks independently.
- The latest deadline used by a job d_k^s is an upper bound of its finishing time, therefore:
  - $\delta_k = d_k^s - r_k$ is an upper bound of the response time. δ_k is an overapproximation of the deadline;

  - The probability of a deadline miss is $Pr\{\delta_k \leq N \cdot T_s\}$

- If we can assume that the task receives only the guaranteed badnwidth, then it is possible to find a model for the evolution of d_k:

$$\delta_{k+1} = \max\{0, \delta_k N \cdot T_s\} + \left\lfloor \frac{c_{k+1}}{Q^s} \right\rfloor T_s$$

Missing Markov-Chain

5.3 Multiprocessor Scheduling

We have built all the theory with a uniprocessor, how much of it can we port to a multiprocessor scheme? There are two option:

- **Partitioned scheduling**: statically assign a task to a processor, that is split the task set $\mathcal{T}$ between the M CPUs. The problem is that the system may be underutilized. In Linux we can request this by the *schedule affinity* option that allows us to schedule a task to a core of the CPU every time.

- **Global scheduling**: in this case there is one single ready task queue and we select the first M tasks from the queue to be distributed on the different M CPUs. This implies allowing *migration*, i.e., the possibility of having one task on a core once and on another, another time. Migrations are actually expensive and the theory for uniprocessor does not hold in some cases for optimality.

6 Linux Scheduling

There are two standard approaches in Linux which are called `SCHED_FIFO` and `SCHED_RR`. They are based on fixed priorities and can be used to implement rate monotonic (RM) and deadline monotonic (DM). The difference between the two appears when two tasks have the same priorities. With the former, tasks are scheduled based on their arriving order, while in the latter they are scheduled based on their rating. Both policies are meant to be general purpose and they aim at having the highest responsiveness preferring real-time tasks to non real-time ones. With `SCHED_OTHER`, we define all *non* real-time tasks.
As a general rule we should try to avoid tasks having the same priorities.
In general, non-real time tasks are scheduled in the background respect to the real-time ones. This is not a really good choice since real-time tasks may end up taking all the resources from the non real-time ones and starving them. In order to avoid this, their computation must be limited and the use of real-time priorities should be limited to only when it is actually needed. Moreover, real-time tasks should require root privileges.
Linux provides the *real-time throttling* mechanism which is a way to avoid the possibility of a real-time task being executed too many times or for too long. The use of this mechanism is troublesome since it heavily interfere with the real-time priorities.

6.1 Earliest Deadline First (EDF) and CBS

With EDF, every task is assigned with a priority which is proportional to its absolute deadline. The problem is that the kernel is not aware of tasks deadline, so we need a more advanced API, which needs to assign relative deadlines D_i to the tasks.
To make the kernel aware, we need to notify it about the jobs that are taking place and to do so we need the applications to be modified in order to signal the beginning and the end of a job via system calls. Another possibility would be for the scheduler to assume that a new job arrives each time a task wakes up. Notice that the problem with using system calls is that there are some cases in which the task can go into a blocked state without actually ending a job, for example when it needs to wait for a blocked resource.
Another possibility is to assign *dynamic scheduling deadlines*, which do not always correspond to the deadlines of the tasks, but they are used to determine the execution schedule. If the scheduling deadline matches the absolute deadline of a job, then the scheduler can respect the absolute deadline.
We could do this by using a CBS which can be used to assign a dynamic scheduling deadline d_i^s to a task τ_i by reserving a maximum runtime Q_i every period P_i.
Moreover, the usage of CBS allows also temporal isolation between the task by definition: the worst case finishing time for a task does not depend on the other task since it is guaranteed to receive its reserved time. This solves the issues of real-time task trying to consume too much execution time. If the task is executing for more than reserved, then it's going to be postponed. To do this we can use throttling or decreasing the real-time priority.

We know that CBS is based on CPU reservations (Q_i, P_i), i.e., if task τ_i tries to run for more than Q_i every P_i, the algorithm decreases its priority or throttles it. In this way, τ_i has the same CPU utilization of a task with WCET Q_i and period P_i.
Using EDF on the scheduling deadlines allows to guarantee that τ_i receives Q_i time units every P_i ones if when we allocate the bandwidth Q_i/P_i, the total bandwidth is smaller or equal to 1:

Definition 6.1

τ_i is guaranteed to receive Q_i time units every P_i if $\sum_j \frac{Q_j}{P_j} \leq 1$

Suppose we allocate a sufficient budget, then we know that EDF is optimal on uni-processors or partitioned systems, but what happens when we have a multi-core processor with M CPUs/-cores? In this case, even if $\sum_j \frac{Q_j}{P_j} \leq M$, we cannot guarantee the deadlines, but we can only guarantee each task to receive Q_i every P_i with a maximum delay.
CBS allows to support self-suspending tasks, so that we don't have to strictly respect the rate monotonic scheduling model (Liu&Layland model) and we don't need to explicitly signal jobs.
Let's quickly recap how the CBS algorithm works:

- Each task τ_i is associates with a scheduling deadline d_i^s and a current runtime q_i, both initialised to 0 when the task is created.
- When a task wakes up, it checks if the current scheduling deadline can be used $d_i^s > t$ where t is the remaining time for the server and $q_i/d_i^s - t < Q_i/P_i$. If not then a new deadline is set and the capacity of the server is recharged:

$$d_i^s = t + P_i, \quad q_i = Q_i$$

- τ_i executes for a time δ and when it finishes, we update the server remaining capacity: $q_i = q_i - \delta$.
- When the server capacity runs out $q_i = 0$, τ_i cannot be scheduled until time d_i^s, at which moment, a new deadline is chosen and the server capacity is updated:

$$d_i^s = d_i^s + P_i, \quad q_i = q_i + Q_i$$

Once the CBS has been used to assign scheduling deadlines to the tasks, then EDF can be used to schedule them. This combination is called `SCHED_DEADLINE` in Linux.
With multiple processors we have some options:

- We can choose to use a partitioned scheduling, i.e., assign each task to a CPU in a fixed way and use the same scheduling logic on each CPU.
- We can choose to allow migration and thereby adopt a global scheduling solution.

Both of these solutions are available in Linux and are selectable by using `cpuset` mechanism to chose one or the other.
When using `SCHED_DEADLINE`, we need to decide how to set the scheduling parameters (Q_i, P_i). The most important thing is that we need to make sure that we don't violate the idea that the cumulative allocation of the bandwidth, which must be less than the number of the processors:

$$\sum_i = \frac{Q_i}{P_i} \leq M$$

The kernel checks the tasks when admitting them (**admission control)**.
`SCHED_DEADLINE` can serve both:

- *Hard real-time tasks*: if $Q_i \geq C_i$ and $P_i \leq T_i$, that is, if the maximum runtime is larger than the WCET and the server period is smaller than the task period, then the scheduling deadlines are equal to the jobs' deadlines. In this case, with uniprocessors and partitioned systems, all deadlines are guaranteed to be respected, while in case of global scheduling an upper bound for the tardiness is provided.

- *Soft real-time tasks*: if $Q_i < C_i$ or $P_i > T_i$, then we require that ${}^{Q_i}/_{P_i}$ must be larger than ${}^{\overline{c_i}}/_{\overline{t_i}}$ otherwise the deadline of the task tends to infinite and there will be no control on the task's response time. Soft tasks allows to do also adaptive bandwidth allocation: suppose we don't know anything about the computation time or the activation time of a task. What we can do is to try by assigning some values of Q_i and P_i and then, by monitoring the deadline misses, we can change the values according to what we are seeing. The logic is based on stochastic analysis: given $c_i < Q_i < C_i$, $T_i = nP_i$, and the probability distributions of execution and inter-arrival time, then it's possible to find the probability distribution of the response time (and the probability to miss a deadline).

7 Kernel

An operating system can be defined as:

- A set of computer programs;
- An interface between applications and hardware;
- The controller for the execution of application programs;
- The manager for the hardware and the software resources. In this case we can see the operating system as:
  - A service provider for user programs exporting a programming interface;
  - A resource manager implementing schedulers.

Using the kernel services a task has the illusion of running on a reserved processor. We can do this by using two important components:

- Scheduler: decides which task to execute at a time;
- Dispatcher: actually switches the CPU context for the task.

There are also programming paradigms provided by the kernel that allow the tasks to communicate:

- Shared memory (threads): the kernel provides mutexes and condition variables with real-time resource sharing protocols;
- Message passing (processes): the kernel provides pipelines and client/server interactions model. Moreover it provides Inter-Process Communication (IPC) mechanisms like messages queues and others. From the real-time prospective, priority inversion is a problem, so we still need real-time protocols;

An adequate scheduling of system resources removes the need for over-engineering the system and is necessary for providing a predictable Quality of Service (QoS). We use real-time protocols to remove the need of over-engineered processors.
The theory has provided us with nice algorithms, but the implementation is another story: real-time operating systems not only need to have the algorithms written, but they have to correctly implement them. The main problem is the necessity of using accurate timers: indeed, if the timers are not correct, the activation time of a task drifts away and a periodic task becomes less and less periodic. Moreover, suppose that a task becomes available, we want the scheduler to be able to notice the task as soon as possible. The more the task waits, the more it is blocked. Finally, the context switch must be as fast as possible, requirement that will take the name of the *dispatcher problem*.
Let's consider a periodic task:

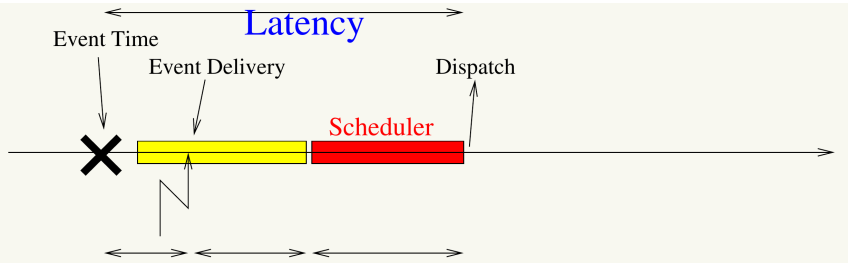
```c
while(1){
  /*Job body*/
  clock_nanosleep(CLOCK_REALTIME, TIMER_ABSTIME, &r, NULL);
  timespec_add_us(&r, period);
}
```

Such task expects to be activate at time $r = r_0 + jT$, but the reality is that may be delayed: $r = r_0 + jT + \delta$ which can lead to deadline misses.
One of the main reasons for such delay is known as *kernel latency*, and it can be modelled as a blocking time[1]:

Before	After
$\sum_{k=1}^{N} \frac{C_k}{T_k} \leq U_{lub}$	$\forall i, 1 \leq i \leq n, \sum_{k=1}^{i-1} \frac{C_k}{T_k} + \frac{C_i + \delta}{T_i} \leq U_{lub}$
$R_i = C_i + \sum_{h=1}^{i-1} \left\lceil \frac{R_i}{T_h} \right\rceil C_h$	$R_i = C_i + \delta + \sum_{h=1}^{i-1} \left\lceil \frac{R_i}{T_h} \right\rceil C_h$
$\exists 0 \leq t \leq D_i : W_i(0,t) = C_i + \sum_{h=1}^{i-1} \left\lceil \frac{t}{T_h} \right\rceil C_h \leq t$	$\exists 0 \leq t \leq D_i : W_i(0,t) = C_i + \sum_{h=1}^{i-1} \left\lceil \frac{t}{T_h} \right\rceil C_h \leq t - \delta$

The scheduler is triggered by internal or external events. The time between the triggering event and the dispatch is the *kernel latency*. This is composed of:

- Event generation;
- Event delivery (interrupts may be disabled);
- Scheduler activation, which is a non-preemptable section;
- Scheduling time.



All these different components could introduce long delays. When we move from the theoretical to the real world, we need to take into account these blocking times. What can we do to make it predictable and hence reduce it?
Let's first analyse the components of an operating system:

- One or more processor which can access:
  - *general purpose registers*: contain data or addresses;
  - *special registers*: control the execution of the CPU and hence must be protected, i.e., there are some operations that cannot be done by the user directly, but should be done only by the operating system. For example a user program should not be allowed to influence the CPU mode of operation, nor it should be allowed to perform I/O operations or to reconfigure virtual memory. For these reasons we need a privileged

[1] Recall that U_{lub} is the utilization least upper bound

mode of execution: in this mode, the processor can do things that normally it is not allowed to do and operate on registers on which it normally cannot. The user programs are low privilege level, while the OS kernel runs in *supervisor mode*.

- RAM;
- I/O devices: which are blocks that are interconnected by a communication infrastructure called bus which can have multiple levels.

Some examples of special registers are:

- Program Counter (PC);
- Stack Pointer (SP);
- Flags register;

Let's consider the Intel x86 (32 bit). It has some general purpose registers

- EAX: main accumulator;
- EBX: sometimes used as base for arrays;
- ECX: sometimes used as a counter;
- EBP: stack base pointer;
- ESI: source index;
- EDI: destination index;

and some special registers. The x86 chipset actually uses a segmented memory architecture: the memory is split in different segments based on what they need to do. If we try to access the segment of somebody else, we get an hardware exception. Some of the special registers are:

- CS: code segment;
- DS: data segment;
- SS: stack segment;

This CPU architecture also offers different modes of operation:

- Real mode (RM): maintained to mimic the Intel 8086 addressing mode;
- Protected mode (PM): contains different modes:
  - Virtual 8086 mode: allows the execution of real mode applications that are incapable of running in protected mode;
  - Long mode: used to access 64-bit registers.

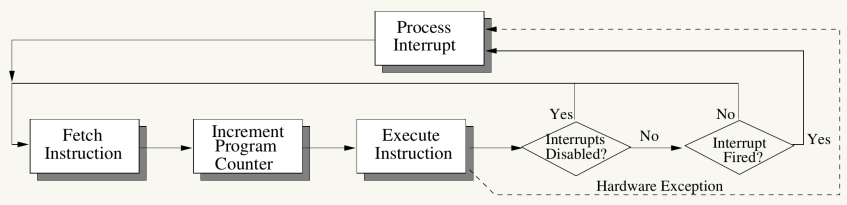
7.1 Kernel

The kernel is the part of the OS that directly manages the hardware, so it runs with the CPU in supervisor mode, also called *kernel level*, and the **kernel space** is composed of the data and the code inside the kernel.
Since there are also some programs that run in user space, mechanisms for increasing the privilege level in a controlled way have been put in place:

- Interrupts like a timer: they are a way to automatically change the mode since the interrupt needs to be managed at a kernel level.

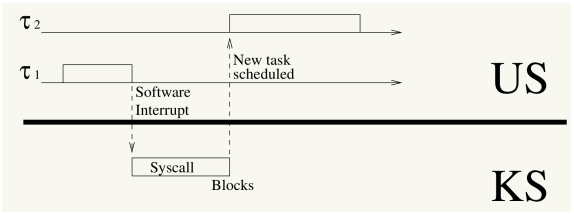
- Instructions causing a hardware exception: these are interrupts that are sent via software (`INT`) and have the same result. An example of software interrupt is the one that is sent when opening a file since it needs to be sent to access the memory.

When an interrupt arrives, a partial context switch is performed: we push the flags and the pointer counter into the stack so that we can resume the execution from the point in which it was suspended. If the interrupt arrives from the user level, we need to switch our stack as well and switch to the kernel stack. After this we jump to the interrupt handler which saves the user registers and restore them later. When we have finished, we go back to the user space through a return from interrupt instruction `IRET`.
To better understand how interrupts work, let's consider the following diagram:



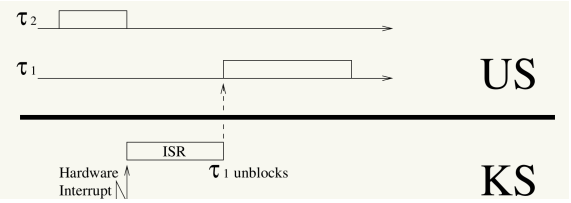
After executing the instruction, if the interrupts are disabled, then the processor starts back, otherwise if they are enabled, the OS checks if an interrupt has been fired. If it has not, then the flow starts back, otherwise the CPU looks into a table to get the correct way to handle the interrupt. The table maintains the addresses of the handlers w.r.t each interrupt associated with an id. Then the handler is executed after switching to the kernels space and having pushed the flags and the PC on the stack.
The handlers table can both be implemented in hardware (Intel) or in software. In the second case, the CPU jumps to a routine which then decides where we should get the handler. From our perspective it doesn't change whether it is hardware or software. The Intel hardware version is called Interrupt Description Table (IDT).
Let's see how software and hardware interrupts are handled:

Software – system call



1. Task τ_1 executes and invokes a system call;
2. Execution passes from the users space to the kernel space, i.e., change of stack, push of the program counter and flags and increase of the privilege level;
3. The invoked system call executes;
4. τ_1 blocks and a new task τ_2 is scheduled.

Hardware



1. While τ_2 is executing, a hardware interrupt fires;
2. The execution passes from the user space to the kernel space;
3. The proper Interrupt Service Routine (ISR) executes;
4. The ISR can unblock τ_1 when the execution returns to the users space and τ_1 is scheduled.

So summing up, the execution enters the kernel for two reasons:

- Reacting to events "coming from up", such as system calls;
- Reacting to events "coming from below", such as hardware interrupts from a device;

Notice that:

- a system call can block the invoking task or can unblock a different task;
- an ISR can unblock a task

The scheduler is invoked when returning from the kernel space to the user space. Since when a task is blocked and unblocked, a context switch may happen, we should try to invoke the scheduler as late as possible.

7.1.1 I/O Operation

Generally, an I/O operation is performed in 3 phases:

- Setup: prepare the device for the I/O operation;
- Waiting: wait for the end of the operation;
- Cleanup: complete the operation.

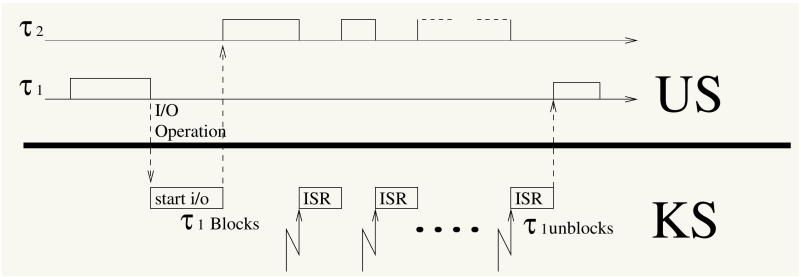
There are different techniques to implement an I/O operation:

- **Polling**: user programs invoke the kernel and the execution is done in kernel space until the operation is not terminated, which can be checked by reading a register. This is not a good idea because we are asking not only to halt the task, but everything. In some cases though, the operation are very small and can be processed very quickly. Polling is done as follows:
  1. The user program raises a software input;

2. Setup phase – in kernel: if the operation is an input operation, then nothing is done, while if it is an output operation, it writes a value to a card register;
3. Wait phase – in kernel: cycles until a bit of the card status register becomes 1;
4. Cleanup – in kernel: if the operation was an input one, then we read the value from a card register, while if it was an output one, then nothing is done;
5. Send `IRET`.

- **Programmed I/O (PIO)** or **interrupt mode**: when the user program invokes the kernel, the execution is moved to the kernel space, and the task that invoked the kernel space is blocked, but the execution retunes to the user space while waiting for the device. The system call activates and enables the blocking. The interrupt handler will take care of notifying the kernel when the I/O operation is ready. The advantage is that we do not waste processor time by waiting for the device, but if frequent short interruptions of unrelated user-space tasks arrive, they can have a significant impact on the execution due to context switches.
  PIO is done as following:

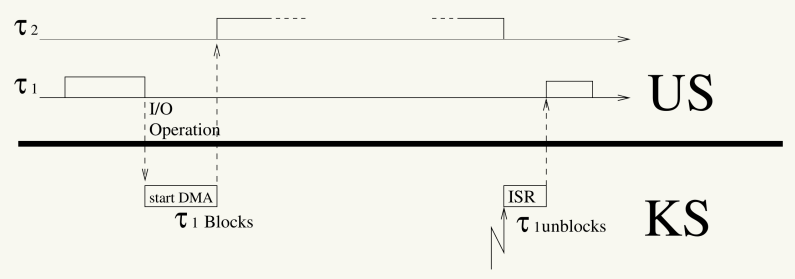
  1. The user program raises a software input;
  2. Setup phase – in kernel: instruct the device to raise an input when it is ready for the I/O operation;
  3. Wait – in user space: block the invoking task and schedule a new one;
  4. Cleanup – in kernel: the interrupt fires, the instructions are moved to the kernel space and then the I/O operation is performed;
  5. Return to phase 2 or unblock the task if the operation is terminated.



- **Direct Memory Access (DMA)** or **bus mastering**: the I/O operations are delegated to the peripheral itself: the user program invokes the kernel, the execution returns to the user space while waiting for the device and the task that invoked the system call is blocked. The I/O operations are not performed by the kernel on interrupts, but by a dedicated hardware device and an interrupt is raised when the whole I/O operation is terminated. This is done when we want to transfer large amount of data. The problem is that there is a cost: when the task is blocked, we are not using processor time, but the bandwidth of the bus, so if the processor needs to access it, there may be problems.
  DMA is done as follows:

  1. The user program raises a software input;
  2. Setup phase – in kernel: instruct the DMA, or the bus mastering device, to perform the I/O operation;
  3. Wait – user space: block the invoking task and schedule a new one;

4. Cleanup – in kernel: the interrupt fires since the operation is terminated;
5. Unblock the task and invoke the scheduler.



7.2 Timer Latency

Latency is the difference between the theoretical scheduled time and the actual time scheduled time. A task τ is expected to be scheduled at time t, but it is actually scheduled at time t'. The latency is computed as $L = t' - t$.
The latency L can be modelled as a blocking time which affects the guarantees test. This is similar to what happens with shared resources, but since the blocking time is due to latency, no priority inversion is involved.
What we can do is to try to identify an upper bound for the latency:

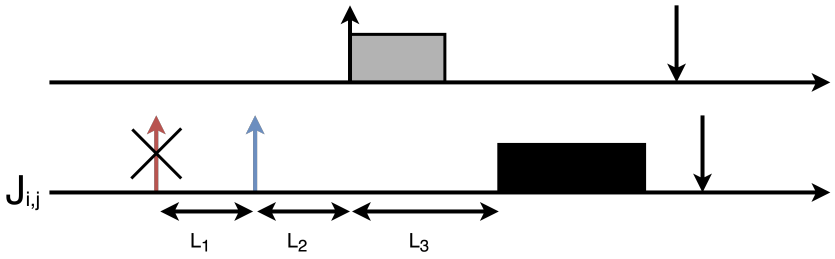
$$\exists L^{max} : L < L^{max}$$

If such upper bound is not known, then we cannot do any schedulability test. Moreover, if L^{max} is too high, only a few task sets result to be schedulable since all tasks will experience large blocking time.
A task τ_i is a stream of jobs $J_{i,j}$ expected for time $r_{i,j}$. Consider a job $J_{i,j}$ which arrives at time $t' > r_{i,j}$, then the latency can be computed as:

$$L_{J_{i,j}} = t' - r_{i,j}$$

There are three components to L:

- L_1 is the delay due to the delay of the interrupt which is not immediately propagated neither to the task nor to the kernel. It's called **interrupt generation latency** L^{int}.
- L_2 the system reaches the kernel level. In this moment, some operations avoid preemption generating a non-preemptable section. It's called **non-preemptable section latency** L^{np}.
- L_3 is scheduling interference and it is perfectly fine since we account for it with the scheduling analysis. It's called **scheduler interference** and is the interference from the higher priority tasks.



The non-preemptable section latency is the time that passes between the generation of an event and the time the kernel finishes handling the interrupt. This is mainly due to non-preemptable sections in the kernel which delay the response to the hardware interrupts. This is the part in which we can work the most since a smart structure of the kernel can diminish such latency. We'll see it better later.

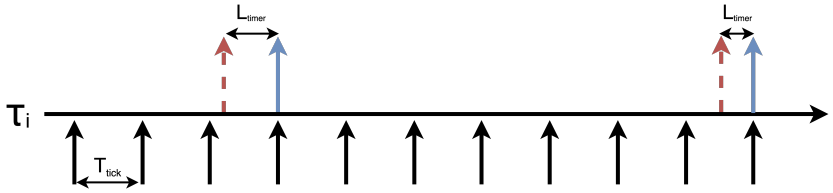
7.2.1 Timer Resolution Latency

L^{int} is small compared to L^{np} and is a time due to the hardware generating the interrupt. We cannot work a lot on this latency, but there is an exception: when the device is a timer device, the interrupt generation latency cannot be high otherwise due to drifting we would get wrong values. We shall call this **timer resolution latency** L_{timer}. Such latency comes from the fact that kernel timers are generally implemented using a hardware device that produces periodic interrupts. Let's consider a periodic task τ_i with a timer which needs to activate the task with a period T_i. The timer sent a periodic interrupt called `tic` with period T^{tic} and if T_i is not a multiple of T^{tic}, then it needs to wait and it can drift apart.
At every `tic`, the execution enters the kernel space and it can, for example:

- Wake up tasks;
- Adjust tasks priorities;
- Run the scheduler when returning the user space which can lead to possible preemption.

There is an evident trade-off: with a timer tick that is too large, we could have resolution problems since the timer is called with a too large delay. Instead a small timer tick means a higher number of interrupts, which leads to more context switches and a large overhead. Reducing the timer tick below 1ms is generally not acceptable.
The timer resolution latency is experienced by all the tasks that want to sleep for a specific time T_i. Indeed, even if a task τ_i wants to wake up at time: $r_{i,j} = jT_i$, the reality is that it will be woken up at time:

$$t' = \left\lceil \frac{r_{i,j}}{T^{tic}} \right\rceil T^{tic}$$



It's possible to show that the timer resolution latency is actually bounded; let $t = r_{i,j}$, then:

$$\begin{aligned} L_{timer} = t' - r_{i,j} &= \left\lceil \frac{r_{i,j}}{T^{tic}} \right\rceil - r_{i,j} \\ &= \left(\left\lceil \frac{r_{i,j}}{T^{tic}} \right\rceil - \frac{r_{i,j}}{T^{tic}} \right) T^{tic} \leq T^{tic} \end{aligned}$$

From the previous image, we can notice that a lot of useless timer interrupts may be generated.

7.2.2 Timers and Clocks

A timer is used to generate an event at a specified time t, while a clock is used to keep track of the current system time. So a timer can actually be used to wake up a task τ, while a clock can be used to read the system time.
We can define the **timer resolution** as the minimum interval at which a periodic timer can fire, if periodic ticks are used, then the timer resolution is T^{tic}.
Instead, we can define a clock resolution as the minimum difference between two different times returned by the clock. In traditional operating systems, a tick counter is used as a clock: it's a very fast clock and basically computes the time by returning the number of ticks since the system boot. In this case the clock resolution is T^{tic}. Modern PCs implement higher resolution time sources in hardware, such as the TimeStamp Counter (TSC) from Intel x86.
Unfortunately this is not good to create a timer since we cannot check every time what the value is.
A timer is usually hardware implemented and it is a device that has a register in which the counter value C is decremented and when it reaches 0, then an interrupt is sent. A timer device generally works in 2 modes:

- **Periodic**: the counter register is automatically reset to the programmed value;
- **One-shot**: the kernel has to explicitly reprogram the device setting the counter register to a new value.

The periodic mode is easier to use since it does not need any kernel intervention. On the contrary, when using the one-shot mode, we introduce a delay which goes accumulating with the risk of drift between real time and system time. This is due to the fact that the timer interrupt handler must:

- Acknowledge the interrupt handler;
- Check if a timer expired;
- Compute when the next timer must fire;
- Reprogram the timer device to generate an interrupt at the correct time.

The last two points are particularly critical and problematic. Indeed, when the kernel reprograms the timer device it must know the current time, but the last known time is the time when the interrupt fired (before step 1). A free run counter (which is not stopped at time when the handler is started) is needed so that the counter is synchronized with the timer device.

7.2.3 Non-Preemptable Section Latency

It is given by the sum of different components:

- *Interrupt disabling*: physical problem due to the fact that we want to guarantee data consistency;
- *Delayed interrupt service*: related to the first one due to the need for maintaining consistency;
- *Delayed scheduler invocation*: every time we make a change, e.g., serve an interrupt, it can happen that some tasks change the priority so we should call the scheduler. It is also possible though that some other tasks in the near future may change the priorities again, so we can delay the call to the scheduler a little bit to minimize the number of preemptions and increase the system throughput.

Protected instructions are instructions that can be executed only in kernel mode. In modern system, only the kernel can enable or disable the interrupts, which is done to ensure consistency. When the interrupt is fired, the Interrupt Service Routine (ISR) is run. They are usually small and do only a few things, such as setting some kind of software flag to notify that the interrupt fired. Later the kernel can check such flags and run a larger interrupt handler. This increases the delay.
It's possible to distinguish between two different kinds of ISRs:

- Hard Interrupt Request (IRQ) handlers:
- Soft Interrupt Request (IRQ) handlers: they can re-enable hardware interrupts and the action of enabling/disabling them is simpler and cheaper. The main problems are the fact that they increase the non-preemptable latency and that they are often non-preemptable increasing the latency for other tasks too. Another main aspect to be taken into consideration is the *deferred scheduling*, i.e., the scheduler is invoked when returning from the kernel space to the user space, but sometimes it returns to the user space after a lot of activities. We need to try to reduce the number of context switches from kernel space to the user space: this would increase the throughput but at the same time it would worsen the latency performance.

Summing up, the non-preemptable latency depends on different factors but none of them is an hardware reason. Indeed, it almost entirely depends on the kernel structure: it is generally the result of the strategy used by the kernel for ensuring mutual exclusion on its internal data structures.
Some factors are:

- Interrupt handling;
- Management of parallelism.

7.3 Real-Time Executives

The main reason because we have problems is the operating system. We could smartly choose the operating system in order to obtain the better performance for real-time tasks.
Real-time Executives are more a library than an operating system. No difference between user space and kernel space. The applications are allowed to execute privileged instructions. This is a little bit extreme since, instead of using an OS that has all the protections, we can use this library that gives the minimum support. The advantages are that is simple, small and with a low overhead. Moreover we link only the specified pieces of the kernel that we are interested in. An example is FreeArtos or AutoSar. The last one provides some sort of protection but at the same time it tries to retain the same advantages of real-time executives.
The way we guarantee mutual exclusion is by disabling interrupts, which all the tasks can do. The non-preemptable latency L^{np} is bounded by the maximum amount of time the interrupts are disabled.

7.4 Monolitic Kernels

This are the opposite of real-time executives. They have protection by having a clear distinction between the kernel and the user applications. The kernel behaves as a single-threaded program, which allows to simplify the consistency of internal kernel structures. The execution enters the kernel in two ways: coming from upside, that is by a system call, or coming from below, that is

from a hardware interrupt.
Since it's not possible to execute more than a flow in the kernel, it is not possible to execute more than 1 system call at time. When considering Symmetric Multi-Processing (SMP) systems, we create a critical section for each system call so that they execute in mutual exclusion.
In Linux, the interrupt handler is split in two parts in order to compromise both speed, the handler should end fast in order to not keep interrupts blocked for too long, and the possibility to have a lot of work to be done to handle an interrupt.

- **Top-half handler**: it's a short and fast Interrupt Service Routine (ISR) in order to minimize the time in which the interrupts are disabled.
- **Bottom-half handler**: it's a soft Interrupt Request (IRQ) handler, which is scheduled by the top-half some time after the top-half has finished responding to the interrupt. In this way, the bottom-half handler can work a in a safer environment, i.e., interrupts are not disabled. A bottom half is always called at the end of a system call before invoking the scheduler for returning to the user space. Moreover, bottom halves are executed sequentially so there is no problem about the consistency. The problem is that we reduce the parallelization of the kernel, which can be bad for the real-time aspect.

As an example, a top-half handles an interrupt for storing data in a device, by first saving it to a device-specific buffer, then scheduling a bottom-half and exiting. This is very fast, then the bottom-half performs whatever other work is required, such as awakening processes, starting up another I/O operation and so on. This setup permits the top-half to service a new interrupt while the bottom-half is still working.
Kernels working in this way are often called *non-preemptable kernels* and the L^{np} is upper-bounded by the maximum amount of time spent in kernel space.
How are monolithic kernels supposed to run on multiprocessors since they are single-threaded? The traditional solution used to ensure consistency are spin-locks. How do we remove the need for this big kernel lock which is something that is bad both for real-time systems and throughput? One thing that can be done is to split the big lock into multiple smaller locks: indeed, we don't have to lock everything all the time since there are some cases in which we are not using global data structures, but only local ones. The idea then is how to make a better split between sections that need to be executed in mutual exclusion. This means having more parallelism in the kernel, but the tasks in kernel space are still non-preemptable.

7.4.1 Preemptable Kernels – Non-Preemptable Protocol

The most recent kernels have a multithreaded kernel: fine-grained critical sections inside the kernel which is still non-preemptable. The idea is that we check for preemption when exiting the kernel's critical sections. To do this check we need to modify the spin-lock and introduce a counter: when a task enters the critical region increments the counter and checks the lock, then when it exits, it checks the counter to see if someone is waiting. The critical section is still non-preemptable, so L^{np} is upper bounded by the maximum size of a kernel critical section.
The preemptable kernels use non-preemptable protocol for kernel critical sections. This is known to have some issues: low-priority tasks with large critical sections can affect the schedulability of high-priority tasks not using resources; low-priority tasks invoking long system calls can compromise the schedulability of high priority real-time tasks.
Alternatives are: Highest Lock Protocol (HLP) and Priority Inheritance Protocol (PIP). HLP is easy to implement, but requires to know which resources the tasks will use. On the contrary, PIP does not require a-priori knowledge of the tasks behaviour, but it requires more changes to the kernel.

7.4.2 Highest Lock Protocol (HLP)

We make a distinction between real-time tasks (do not use the kernel) and non real-time tasks which can use the kernel. In this way, a real-time task would become "useless", but there are two solutions:

- micro-kernels;
- dual-kernels.

7.4.2.1 micro-kernel

The idea is to simplify the kernel: reduce the number of system calls and the interaction is done via IPC. All the kernel needs to do is to implement:

- Address spaces
- threads
- IPC mechanism

While it does not need to do:

- Virtual memory management
- Scheduler
- Interrupt handlers

which are done as a service at user level.
The different kernel services are implemented by user space processes. We could have a single-server OS that is an OS kernel implemented as a single user-space process and also multi-server OS that is an OS kernel implemented as a multipl user-space process.
Unfortunatelly, the latency reduction achieved by the micro-kernel, is often not sufficient for real-time systems. Also in this case we need to introduce preemtability as in monolitic kernels. The problem with micro-kernels is the fact that we have a lot of switches since the messages need to be sent in kernel space. We need to reduce the interprocessor call preemptable or reduce the number of switches.
A second generation of micro-kernels is called L4 and it was born with the idea is that, if it wants to use for real-time, the microkernel must be small:

- Very simple kernel;
- Small
- Super-optimized IPC: designed to be efficient and not powerful

L4 was also combined with Linux by binding the Linux kernel with the IPC mechanism instead with the hardware directly. In this way, non real-time tasks could be run on Linux and real-time tasks could be run on L4.
Unfortunately the performance were not so good, and they had to rewrite another kernel to provide low latencies which is called Fiasco.
In L4Linux they tried to disable the interrupts: they remapped to a soft interrupt disabling, that is, instead of having the CLI to really disable the interrupt, to a soft one that simply suspend the notifying of the interrupts on IPCs and then when reenables them, you find all of them???

7.4.3 Dual-kernel

The idea is to take the best from real-time executives and monolitic kernels.
In this way we have a lower level real-time kernel that manages hardware interrupt and forwards the interrupt to Linux (monolitic kernel) only when they do not mess up the real-time activities. In this way, we cannot have the case in which Linux can stop interrupt to disable interrupts directly.
The first solution was called real-time Linux (RTLinx): patch for Linux kernel to intercept the interrupts.
RTLinux was patented, then RTAI came from a community effort. Real-time applications do not forward interrupt but execute directly inside the kernel. The idea came from an old paper talking about interrupt pipelines: a small nanokernel handles interrupts by sending them to pipelines of applications/kenrel that actually manage them. Real-time applications come first in the pipeline.

7.5 Real-time in Linux User Space

In this case we want to modify the Linux kernel directly so that it supports real-time applications in user space without the need of complex solutions.
People identified some requirements:

- Fine-grained locking
- Preemptable kernel: it manages the interrupts;
- Schedulable ISRs and BHs
- Replace spinlocks with muxes, complex to be used with multiprocessors;
- A real-time synchronization protocol (PI) for these mutexes.

Acronyms

API Application Programming Interface. 63

CBS Constant Bandwidth Server. 2, 57–60, 63, 64

DM Deadline monotonic. 14, 26

DM deadline monotonic. 45, 63

DMA Direct Memory Access. 72

EDF Earliest Deadline First. 1, 2, 45, 47, 57–60, 63, 64

FCFS First Come First Served. 9, 10

FIFO First In First Out. 51

HLP Highest Lock Protocol. 2, 77, 78

IDT Interrupt Description Table. 70

ILP Integer Linear Programming. 25

IPC Inter-Process Communication. 31, 67, 78

IPCP Immediate Ceiling Priority Protocol. 40–42, 44

IRQ Interrupt Request. 76, 77

ISR Interrupt Service Routine. 71, 76, 77

MIT Minimum Inter-arrival Time. 7, 49, 50

NPP Non Preemptive Protocol. 1, 32–34

OPCP Original Priority Ceiling Protocol. 40–43

PC Program Counter. 69

PCP Priority Ceiling Protocol. 1, 40

PIO Programmed I/O. 72

PIP Priority Inheritance Protocol. 1, 35, 36, 38–40, 77

QoS Quality of Service. 67

RM Rate monotonic. 14, 26, 46, 54

RM rate monotonic. 14, 16, 39, 45, 63

RTA Response Time Analysis. 49

RTA response time analysis. 45, 47

SMP Symmetric Multi-Processing. 9, 77

SP Stack Pointer. 69

TDA Time Demand Analysis. 49

TDA time demand analysis. 45, 47

TSC TimeStamp Counter. 75

WCET Worst-Case Execution Time. 1, 2, 6, 13, 15, 16, 18, 25, 49, 50, 60, 64, 65

WCET worst-case execution time. 19, 52

WCRT Worst-Case Response Time. 6