145071 - Real time operating systems and middleware

Marco Peressutti
230403
marco.peressutti@studenti.unitn.it

January 29, 2023

Contents

Introduction to the Course

Material

- Slides available from moodle
- Interested students can have a look at: *Giorgio Buttazzo*, **HARD REAL-TIME COMPUTING SYSTEMS: Predictable Scheduling Algorithms and Applications**

Exam

Written Exam

- 3 questions, 30 minutes per question
- Each answer gets a score from 0 to 30
- (Optional) project

Oral Exam

- Discussion of the written exam
- Open Questions or Discussion on a project

Prerequisites

- Programming skills: C, maybe C++.
  You must know how ot code in C (optionally C++). This is not about knowing the C syntax, it is about writing good and clean C code.
  To help overcome this lack of prerequisites please consider reading the book *Kerrigan & Ritchie*,**The C Programming Language**
- Knowledge about Operating Systems.
  This prerequisites is met if you have taken the course *Sistemi Operativi 1* or similar exams.
  Alternatively please refer to a good Operating Systems book (e.g. Stallings,. . .).
  This includes how to use a shell, basic POSIX commands, `make`, how to compile, ...

Overview of the Course

The course will cover 6 main macro areas of real time operating systems and middleware.

1. Real Time Systems:

    - Real-Time Computing and temporal constraints.
      Real time systems are software and hardware systems (hence computing systems), that have to comply with temporal constraints.
    - Definitions and task model
      We will make things much clearer and better defined by introducing a sequence of definitions and mathematical models that will allow us to given this notion of temporal constraint a well founded meaning.

- Real-Time Scheduling
  We will also study solutions that allow us to enforce these real time constraints and this solution will have much to do on how we schedule shared resources.

2. Real-Time programming, RT-POSIX, pthreads,...
   We will move to a concrete ground and see what is the exact shape that these notions take once they are moved in a computer program.
3. Real-Time Scheduling algorithms:
   - Fixed Priority scheduling, RM, DM
   - EDF and dynamic priorities
   - Resource Sharing (Priority Inversion, ...)

   As regards the Real-Time scheduling we will see many interesting policies, but since this is not a course on Real Time scheduling what we will do is provide the knowledge of real time scheduling so that the reader will be able to understand the mechanism of real time operating systems and thereby make best use of these technologies in future projects.
4. Operating System Structure
   - Notes about traditional kernel structures
     In order to keep latencies in check, we need proper technological solutions that make our operating systems differ quite a bit from standard operating systems.
   - Sources of kernel latencies
   - Some approaches to real-time kernels (e.g. dual kernel approach, interrupt pipes, micro-kernels, monolithis kernels and RT)
5. Real-Time Kernels and OSs.
6. Developing Real-Time applications

Real-Time Operating Systems

In order to discuss about the Real-Time systems we need to provide some basic definitions:

Definition 1: Real-Time Operating Systems (RTOS)

Operating Systems that provide support to Real-Time Applications

Definition 2: Real-Time application

the correctness depends not only on the output values, but also on the time when such values are produced

Definition 3: Operating Systems (OS)

- Set of computer programs, of critical programs to be precise: because they have to be written efficiently, otherwise the hardware resources get disrupted, hence the system cannot operate correctly.
- Interface between applications and hardware.
  Whenever an application interacts with an hardware, it is not of the developer interest to directly control the hardware. The Operating System provides an API that enables you to open a connection to a peripheral and takes care of all the low level interactions. On this regard, understanding the notion of interrupt will be of fundamental importance, because it is, essentially, what gave rise to concurrent programming: in the case we would like to interact with a peripheral, rather than continuously check if the peripheral has ended what it is supposed to do, you can tell the peripheral to communicate when

it has completed the given task.
Anyway the Operating systems acts as an interface towards the hardware and hides away all these complex details.

- Control the execution of application programs
- Manage the hardware and software resources

Since the Operating System is something that lies in-between the user application and the hardware resources we can summarize the aforementioned interpretation of

- **Service Provider** for user programs (i.e. exports a programming interface). Service Provider
  This concept looks at the OS from the perspective of the software application, in the sense that the Operating Systems provides to the application a series of services:
  - Process Synchronization mechanism
  - Inter-Process Communication (IPC)
  - Process/Thread Scheduling, i.e. ways to create and schedule tasks
  - Input/Output
  - Virtual Memory

  And all these services are accessible through an API.
- **Resource Manager** Resource Manager
  If you think at the Operating System as a Resource Manager, then it is something that takes care of many things:
  1. **Process Management** Process Management
     The fact that multiple applications can run at the same time on a PC, even though there is a small amount of processor available to manage these applications. (generally 2,4 or 8).
     The number of application that you are likely to create is often on the hundreds, hence it is necessary to make an appropriate sharing of the limited resources that you have in order for all the applications to live correctly.
  2. **Memory Management** Memory Management
     Supposing one is using a 64-bit architecture, what will happen is that a space of memory is addressable with 64 bit. As a consequence we can imagine that the addressable memory is space has $2^{64}-1$ memory locations available.
     And each application sees, these much space available for its execution. But however large the space can be in a machine, it will never match the aforementioned size. It could potentially for one task, but in the case a machine is hundreds of tasks and each of them wants to use that much memory, there is no way that the hardware can provide enough physical memory to satisfy all of them.
     To counteract this problem, it is common practice to schedule the memory as well, because you take advantage of the fact that an application CAN use $2^{64}-1$ memory locations, but at a given time it uses a tiny portion of these locations. It is only that tiny portion of memory locations that needs to be made available to the running task.
     In this scenario, the OS makes it possible to accommodate within the physical memory of the computer these small slices of the available space that the application uses. So somehow it operates as a resource manager for the memory as well.
  3. **File Management** File Management
  4. **Networking**, **Device Drivers**, **Graphical Interface** Networking Device Drivers Graphical Interface

  The important thing is that all of these resources, like the processor, the memory, the drivers etc..., are shared between all the tasks. All these resource managers have to be distributed among all the spectrum of tasks in such a way that the tasks behave properly, i.e. if you do not provide frequently enough these resources they would not be able to deliver the result on time (the OS manages this problem on its own).

In the case we decide to look at the Operating System as a Resource Manager, we need to think of a structure for the OS that makes this resource management effective, effective in the sense that we believe it is the most relevant for our specific range of application.
The way OSs handles devices, interrupt, etc. can be very different (and optimized in very different ways) depending on the type of application one is looking at. However, the type of optimizations we are interested in are those that allow our application to have time-limited execution.

Real-Time Systems

A **Real-Time application** is an application of which the time when a result is produced matters. Real-Time application
In particular:

- a correct result produced too late is equivalent to a wrong result, or to no result.
- it is characterized by temporal constraints that have to be respected.

Example 1: Mobile vehicle

Let us consider a mobile vehicle with a software module that

1. Detects obstacles
2. Computes a new trajectory to avoid them
3. Computes the commands for engine, brakes,...
4. Sends the commands

If you decide to steer to the left or to the right there is a limited amount of time in which the operation has to be carried out. Hence if one can find an extremely effective strategy for steering the wheels but the strategy amounts to setting the values for the motors after one second, it is completely useless, since the vehicle is most likely to crash.
Hence a time violation in executing a task is a critical problem: it means that the developed application is useless and also dangerous.
But then, what is a reasonable time frame for completing the steering operation?
Depends on the speed in which the vehicle is traveling. But no matters if the vehicle is traveling at high or low speed the timing constraint is there, and if it is violated, the vehicle will eventually crash against the obstacle.
As a consequence: when a constraint is set, that constraint needs to be respected. And this is one of the core concept of Real Time.

Hence, a Real-Time is not necessarily synonym of fast execution, but rather of **predictable** execution. predictable
Real time computing has much more to do with predictability than of being quick.
Some examples of temporal contraints are:

- The program must react to external events in a predictable time
- The program must repeat a given activity at a precise rate
- The program must end an activity before a specified time

In this case, we can clearly notice that the temporal constraints can be either one shot events or periodic events, but in both cases, a common characteristic, there is a need of being predictability.
Temporal constraints are modeled using the concept of **deadline**. deadline
Please notice that a Real-Time system is not just a *fast system*, because the speed is always relative to a specific environment, i.e. the steering commands temporal constraint is set by the velocity of the vehicle.
Running faster is good, but does not guarantee the correct behavior. In fact, it is far more valuable to that temporal constraints are always respected; in other terms Real time systems prefer to run fast enough to respect the deadlines, to be reliable.

Hence, the type of analysis that is necessary to perform is not an analysis based on of average/typical cases but rather an analysis of worst case: I have to prove that even in the worst-case scenario, there is not deadline violation.

This predictability creates a wide gap between what a Real Time system is and what a general purpose system is, because general purpose systems are optimised for the average case, but a real time system only cares about the worst case. As a consequence, the way one designs a Real Time system is very different from the way a general purpose system is designed.
In fact:

- When one optimize for the average case, what one would look at is the number of times that an application completes a task every second, and this is called **Throughput**. Throughput
- When one have a worst case requirement, the notion of throughput is not relevant anymore, and the analysis focuses in every single instance the maximum delay will be bounded.

Let us introduce some notion and general terms that we will extensively using during the course

Definition 4: Algorithm

Logical procedure used to solve a problem

Definition 5: Program

Formal description of an algorithm, using a *programming language*

Definition 6: Process

Instance of a program (program in execution)

Definition 7: Thread

Flow of execution, something that is able to execute using your processor along with other threads. These threads can be part of the same program and they can be executed in parallel.

Definition 8: Task

Process or thread

Hence there are two different ways of sharing resources: one are threads in which your share computing resources and memory space, and processes in which you share computer resources but each of the processes has its own memory space.
Unfortunately, there is no common definition of a task: somebody use the terms with the same meaning as a thread and sometimes it is used with the same meaning as a process. In this class we will refer to threads.
Henceforth, when we talk about a task we will refer to a program that it is running and they share the same memory space with other programs.

Chapter 1

Basic Concepts

A task can be seen as a sequence of actions and a deadline must be associated to each one of them. We, therefore, are after is a definition of a formal model that identifies what these tasks or actions are and associate deadlines with them.

1.1 Real-Time Tasks

Definition 9: Real-Time Task (τ_i)

stream of jobs (or instances) $J_{i,k}$, or, in other terms, a sequence of activities that is activated periodically or aperiodically

Each job $J_{i,k} = (r_{i,k}, c_{i,k}, d_{i,k})$ is characterised by the following quantities:

- $r_{i,k}$ activation time
  It is the time at which a task becomes ready for execution; it is also referred as *request time* or *release time*.
- $c_{i,k}$ computation time
  Time necessary to the processor for executing the job without interruption.
- $d_{i,k}$ absolute deadline
  time before which a job should be completed to avoid damage to the system.
- $f_{i,k}$ finishing time
  The time at which a job finishes its execution
- $\rho_{i,k}$ response time
  The time at which a job finishes its execution. Formally this quantity is the difference between the finishing time and the activation time.

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

`activation time`
`computation time`
`absolute deadline`
`finishing time`
`response time`

Furthermore, since each task i is a sequence of jobs, we need to differentiate between them. That is why each job $J_{i,k}$ is uniquely identified by its task index i and the k-th activation of the i-th task. In addition, we will say that job $J_{i,k}$ respects its deadline if $f_{i,k} \leq d_{i,k}$.

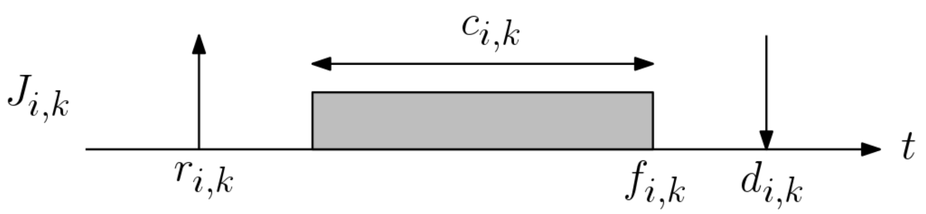


Figure 1.1: Graphical representation of Mathematical model of a Task

This mathematical definition of a job in a real-time task holds regardless of the nature of the task itself. In fact, we can identify three different types of tasks: Periodic tasks, Aperiodic Tasks and Sporadic Tasks. Each of them holds different properties and a different mathematical representation.

1.1.1 Periodic Tasks

Definition 10: Periodic Task

A periodic task $\tau_i = (C_i, D_i, T_i)$ is a stream of jobs $J_{i,k}$, with:

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

where:

- T_i **Period**
- D_i **Relative Deadline**
- C_i **Worst-Case Execution Time (WCET)**
- R_i **Worst-Case Response Time (WCRT)**

Period

Relative Deadline

Worst-Case Execution Time (WCET)

Worst-Case Response Time (WCRT)

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

For the task to be correctly scheduled, it must be $R_i \leq D_i$

A periodic task has a regular structure (called **cycle**), in the sense that:

cycle

- it is activated periodically with a period of T_i
- it executes a computation
- when the computation terminates, it suspends waiting for the next period

Hence, its fundamental implementation can be represented as:

```
void *PeriodicTask(void *arg)
{
  <initialization>;
  <start periodic timer, period = T>;
  while (condition)
  {
    <read sensors>;
    <update outputs>;
    <update state variables>;
    <wait next activation>;
  }
}
```

1.1.2 Aperiodic Tasks

Definition 11: Aperiodic Task

Aperiodic tasks are not characterised by periodic arrivals, meaning that:

- A minimum interarrival time between activations does not exist
- Sometimes, aperiodic tasks do not have a particular structure

Aperiodic tasks can model tasks responding to events that occur rarely (e.g. a mode change) or tasks responding to events with irregular structure (e.g. bursts of packets from the network,...).

1.1.3 Sporadic Tasks

Sporadic tasks are aperiodic tasks characterised by a **Minimum Interarrival Time (MIT)** between jobs. In this sense they are similar to periodi tasks, but while a periodic task is activated by a periodic timer, a sporadic task is activated by an external event. (e.g. the arrival of a packet from the network)

Hence, its fundamental implementation can be represented as:

```
void *SporadicTask(void *arg)
{
    <initialization>;
    while (condition)
    {
        <computation>;
        <wait events>;
    }
}
```

Formally:

Definition 12: Sporadic Task

A sporadic task $\tau_i = (C_i, D_i, T_i)$ is a stream of jobs $J_{i,k}$, with:

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

where:

- T_i **Minimum Interarrival Time (MIT)**
- D_i **Relative Deadline**
- C_i **Worst-Case Execution Time (WCET)**
- R_i **Worst-Case Response Time (WCRT)**

Minimum Interarrival Time (MIT)

Relative Deadline

Worst-Case Execution Time (WCET)

Worst-Case Response Time (WCRT)

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

For the task to be correctly scheduled, it must be $R_i \leq D_i$.

1.2 Task Criticality

A deadline is said to be *hard* if a deadline miss causes a critical failure in the system, whereas a task is said to be a **hard real-time task** if all its deadlines are hard, which means that all the deadlines must be guaranteed before starting the task, i.e.

hard real-time task

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

Example 2: Hard Real-Time Task

The controller of a mobile robot, must detect obstacles and react within a time dependent on the robot speed, otherwise the robot will crash into the obstacles

A deadline is said to be *soft* if a deadlien miss causes a degradation in the **Quality of Service (QoS)**, but is not a catastrophic event, whereas a task is said to be a **soft real-time task** if it has soft deadlines.

Quality of Service (QoS)

soft real-time task

In other terms, some deadlines can be missed without compromising the correctess of the system, but the number of missed deadlines must be kept under control, because the *quality* of the results depend on the number of missed deadlines.

Unline the hard real-time task, soft real-time tasks can be difficult to characterize, particularly:

- What's the tradeoff between *non compromising the system correctness* and *not considering missed deadlines*?
- Moreover, some way to express the QoS experienced by a soft real-time task is needed

Examples of QoS definitions could be

- no more than X consecutive deadlines can be missed
- no more that X deadlines in an interval of time T can be missed
- the **deadline miss probability** must be less than a specified value, i.e.

$$P\{f_{i,j} > d_{i,j}\} \leq R_{max}$$

deadline miss probability

- the **deadline miss ratio** must be less than a specified value, i.e.

$$\frac{\text{number of missed deadlines}}{\text{total number of deadlines}} \leq R_{max}$$

deadline miss ratio

- the maximum **tardiness** must be less than a specified value, i.e.

$$\frac{R_i}{D_i} < L$$

tardiness

- ...

Example 3: Audio and Video players

Assuming a framerate of 25 fps, which imply a frame period of 40 ms, if a frame is played a little bit too late, the user might even be unable to notice any degration in the QoS, however, skipped frames can be disturbing.
In fact missing a lot of frames by 5 ms can be better than missing only a few frames by 40 ms.

Example 4: Robotic Systems

Some actuations can be delayed with little consequences on the control quality.

In any case, soft real-time constraints does not mean no guarantee on dealines, given that tasks can have variable execution times between different jobs.
These execution times might depend on different factors:

- Input data
- HW issues (cache effects, pipeline stalls, ...)
- The internal state of the task
- ...

1.3 Schedulability analysis

Schedulability analysis tries to answer the question: Given a task set $\mathcal{T}$, how can we guarantee if it is schedulable or not?

1.3.1 Simulating the hyperperiod

The first possibility is to simulate the system to check that no deadline is missed. The execution time of every job is set equal to the WCET of the corresponding task.

In the case of periodic tasks with no offsets it is sufficient to simulate the schedule until the **hyperperiod** (H $= lcm\{T_i\}$).

hyperperiod

In the case of offsets $\phi_i = r_{i,0}$ it is sufficient to simulate until $2H + \phi_{max}$.
If tasks periods are prime numbers the hyperperiod can be very large!
In the case of 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 offsets.

1.3.2 (Worst-Case) Response Time Analysis

According to the methods proposed by Audsley et al., the longest response time R_i of a periodic task τ_i is computed, at the critical instant, as the sum of its computation time and the interference I_i of the higher priority tasks:

$$R_i = C_i + I_i$$

where:

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

Hence,

$$R_i = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{1.1}$$

Definition 13: Critical instant

The Critical instant for task τ_i occurs when job $J_{i,j}$ is released at the same time with a job in every high priority task

It is straighforward to notice that if all the offsets of the task set are 0, the first job of every task is released at the **critical instant**. critical instant

A job $J_{i,j}$ released at the critical instant experiences the maximum response time for τ_i:

$$\forall k, \quad \rho_{i,j} \geq \rho_{i,k}$$

No simple solution exists for this equation since R_i appears on both sides of the equation. Thus, the worst-case response time of task τ_i is given by the smallest value of R_i that satisfies equation 1.1. Notice, however, that only a subset of points in the interval $[0, D_i]$ need to be examined for feasibility. In fact, the interference on τ_i only increases when there is a release of a higher-priority task.

To simplify the notation, let $R_i^{(k)}$ be the k-th estimate of R_i and let $I_i^{(k)}$ be the interference on task τ_i in the interval $[0, R_i^{(k)}]$

$$I_i^{(k)} = \sum_{j=1}^{i-1} \left\lceil \frac{R_i^{(k)}}{T_j} \right\rceil C_j \tag{1.2}$$

Then the calculation of R_i is performed as follows:

1. Iteration starts with $R_i^{(0)} = \sum_{j=1}^{i} C_j$, which is the first point in time that τ_i could possibly complete
2. The actual interference I_i^k in the interval $[0, R_i^{(k)}]$ is computed by equation 1.2
3. If $I_i^{(k)} + C_i = R_i^{(k)}$, then $R_i^{(k)}$ is the actual worst-case response time of task τ_i; that is, $R_i = R_i^{(k)}$. Otherwise, the next estimate is given by

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

   and the iteration continues from step 2.

Once R_i is calculated, the feasibility of task τ_i is guaranteed if and only if $R_i \leq D_i$.

The response time analysis is an efficient algorithm: in the worst case, the number of steps N for the algorithm to converge is exponential and it depends on the total number of jobs of higher priority tasks in the interval $[0, D_i]$:

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

If s is the minimum granularity of the time, then in the worst case $N = \dfrac{D_i}{s}$. However, such worst case is very rare, usually the number of steps is low.

1.3.3 Processor Demand Analysis

Another necessary and sufficient test for checking the schedulability of fixed priority systems with constrained deadlines was proposed by Lehoczky, Sha and Ding. The test is based on the concept of Level-i workload, defined as follows

Definition 14: Level-i workload

The Level-i workload $W_i(t)$ is the cumulative computation time requested in the interval $(0, t]$ by task τ_i and all the tasks with priority higher than p_i

The basic idea is very simple: in any interval, the computation demanded by all tasks in the set must never exceed the available time.
The problem is: how to compute the time demanded by a tast set $\mathcal{T}$?
Since we have to look only at jobs released at the critical instant, we can consider all offsets equal to zero and only consider the first job of each task...

Definition 15: Processor Demand

Given an interval $[t_1, t_2]$, let $\mathcal{J}_{t_1,t_2}$ be the set of jobs 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:

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

Worst case: use C_i instead of $c_{i,j}$

Guaranteeing a task set $\mathcal{T}$ based on $W(t_1, t_2)$ can take a long time.
In fact, it must hold

$$\forall (t_1, t_2) \quad W(t_1, t_2) \leq t_2 - t_1$$

This means that the test requires to check all the (t_1, t_2) combinations in a hyperperiod.
However, we only need to check the first job of every task τ_i.
The quantity $W_i(t_1, t_2)$ is the time demanded in $[t_1, t_2]$ by all tasks τ_j with $p_j \geq p_i$ ($\Rightarrow j \leq i$)
We can consider only $W_i(0, t)$.
For task τ_i only check $W_i(0, t)$ for $0 \leq t \leq D_i$.
Change $\forall$ into $\exists$: consider worst case for $W_i()$
The number of jobs in $[0, t]$ is $\left\lfloor \frac{t}{T_i} \right\rfloor$
Use $\lceil\rceil$ instead
We already have hints about computing an upper bound for $W_i(0, t)$...

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

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

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

Sometimes, different notations in literature:

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

This is equivalent, because $0 \leq t \leq T_i$.
Someone defines

$$L_i(t_1, t_2) = \frac{W_i(t_1, t_2)}{t_2 - t_1}$$

$$L_i = \min_{0 \leq t \leq D_i} L_i(0,t) \qquad ; \qquad L = \max_{\tau_i \in \mathcal{T}} L_i$$

The guarantee tests then becomes:

- Task τ_i is schedulable iff $L_i \leq 1$
- $\mathcal{T}$ is schedulable iff $L \leq 1$

The test might still be long (need to check many values of $L(0,t)$ to find the minimum)...
The number of points to check for computing W_i or L_i can be reduced:

$$S_i = \left\{ k\,T_h | h \leq i; 1 \leq k \leq \left\lfloor \frac{T_i}{T_h} \right\rfloor \right\}$$

multiples of T_h for $h \leq i$

$$L_i = \min_{t \in S_i} L_i(0,t)$$

1.3.4 Processor Utilization Factor test

The feasibility of a task set with contrained deadlines could be guaranteed using the utilization based test, by reducing tasks' periods to relative deadlines:

$$U_{lub} = \sum_{i=1}^{n} \frac{C_i}{D_i} \leq n(2^{1/n} - 1)$$

However, such a test would be quite pessimistic, since the workload on the processor would be overestimated.
For this reason this test is **sufficient but not necessary**.

Nonetheless, in many cases it is useful to have a very simple test to see if a task set is schedulable. This sufficient test is based on the **Utilisation bound**. Utilisation bound

Definition 16: Utilisation Least Upper Bound

The utilisation least upper bound for a scheduling algorithm $\mathcal{A}$ is the smallest possible utilisation U_{lub} such that, for any task set $\mathcal{T}$, if the task set's utilisation U is not greater than U_{lub} ($U \leq U_{lub}$), then the task set is schedulable by algorithm $\mathcal{A}$

In other terms, we can consider that each task uses the processor for a fraction of time

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

The total processor utilisation is

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

which we will consider as a measure of the processor's load.

Given these definition, the necessary condition for the schedulability of a task set is:

- If $U > 1$ the task set is surely not schedulable
- If $U \leq U_{lub}$, the task set is schedulable
- If $U_{lub} < U \leq 1$ the task set may or may not be schedulable

Ideally a value of $U_{lub} = 1$ would be optimal.

In general, given that the tasks might not always have relative deadline equals to the period the formulation of the total processor utilisation considers the relative deadline:

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

This approach considers the worst case for a task... hence if the task set is guaranteed using the relative deadlines, it must hold that the test holds even when considering the period.

The bound is very pessimistic: most of the times, a task set with $U > U_{lub}$ is schedulable. A particular case is when tasks have periods that are harmonic.

Definition 17: Harmonic task set

A task set is harmonic if, for every two tasks τ_i, τ_j either T_i is multiple of T_j or T_j is multiple of T_i

For a harmonic task set, the utilisation bound is $U_{lub} = 1$. (Foreshadowing: Rate Monotonic is an optimal algorithm for harmonic task sets)

1.3.5 Examples of schedulability analysis

1.3.5.1 Example 1

Consider a task set of three periodic tasks with deadline equal to period:

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

Now let us consider the schedulability of the task set using the three methods introduced before:

Example 5: Processor Utilization Factor test

First, let us compute U_{lub} for the task set of three tasks with $n = 3$

$$U_{lub} = n(2^{1/n} - 1) = 3(2^{1/3} - 1) \approx 0.77976315$$

Then let us compute the Utilisation for the three tasks:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} = \frac{20}{100} + \frac{40}{150} + \frac{100}{350} = 0.752380952$$

Hence, the sufficient test states that since $U \leq U_{lub}$ the task set is schedulable.

Example 6: Processor Demand Analysis

Let us choose a set of scheduling points for the analysis.

- Task τ_1.

  $$S_1 = \{100\}$$

  For each scheduling point in S_1 find, if any, point for which $W_1(t) \leq D_1$

  $$W_1(0, 100) = \sum_{h=1}^{1} \left\lceil \frac{t}{T_h} \right\rceil C_h = C_1 = 20 \leq 100$$

  Since there exists at least a scheduling point where the relationship is satisfied, τ_1 is schedulable
- Task τ_2.

  $$S_2 = \{100, 150\}$$

  $$W_2(0, 100) = \sum_{h=1}^{2} \left\lceil \frac{t}{T_h} \right\rceil C_h = 20\frac{100}{100} + 40 \left\lceil \frac{100}{150} \right\rceil = 20 + 40 = 60 \leq 100$$

  Since there exists at least a scheduling point where the relationship is satisfied, τ_2 is schedulable
- Task τ_3.

  $$S_3 = \{100, 150, 200, 300, 350\}$$

$$
\begin{aligned}
W_3(0,100) &= 20 + 40 + 100 = 160 > 100 \\
W_3(0,150) &= 2 \cdot 20 + 40 + 100 = 180 > 150 \\
W_3(0,200) &= 2 \cdot 20 + 2 \cdot 40 + 100 = 220 > 200 \\
W_3(0,300) &= 3 \cdot 20 + 2 \cdot 40 + 100 = 240 \leq 300
\end{aligned}
$$

Since there exists at least a scheduling point where the relationship is satisfied, τ_3 is schedulable

Example 7: Response Time Analysis

- Task τ_1.

$$
\begin{aligned}
R_1^{(0)} &= C_1 = 20 \\
R_1^{(1)} &= C_1 = R_1^{(0)}
\end{aligned}
$$

Since $R_1 < D_1$, the task is schedulable.

- Task τ_2.

$$
\begin{aligned}
R_2^{(0)} &= C_2 = 40 \\
R_2^{(1)} &= 40 + \left\lceil \frac{40}{100} \right\rceil 20 = 60 \\
R_2^{(2)} &= 40 + \left\lceil \frac{60}{100} \right\rceil 20 = R_2^{(1)}
\end{aligned}
$$

Since $R_2 \leq D_2$ the task is schedulable.

- Task τ_3.

$$
\begin{aligned}
R_3^{(0)} &= C_3 = 100 \\
R_3^{(1)} &= 100 + \left\lceil \frac{100}{100} \right\rceil 20 + \left\lceil \frac{100}{150} \right\rceil 40 = 160 \\
R_3^{(2)} &= 100 + \left\lceil \frac{160}{100} \right\rceil 20 + \left\lceil \frac{160}{150} \right\rceil 40 = 220 \\
R_3^{(3)} &= 100 + \left\lceil \frac{220}{100} \right\rceil 20 + \left\lceil \frac{220}{150} \right\rceil 40 = 240 \\
R_3^{(4)} &= 100 + \left\lceil \frac{240}{100} \right\rceil 20 + \left\lceil \frac{240}{150} \right\rceil 40 = R_3^{(3)}
\end{aligned}
$$

Since $R_3 \leq D_3$ the task is schedulable.

1.3.5.2 Example 2

Consider a task set of three periodic tasks with deadline equal to period:

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

Now let us consider the schedulability of the task set using the three methods introduced before:

Example 8: Processor Utilization Factor test

First, let us compute U_{lub} for the task set of three tasks with $n = 3$

$$U_{lub} = n(2^{1/n} - 1) = 3(2^{1/3} - 1) \approx 0.77976315$$

Then let us compute the Utilisation for the three tasks:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} = \frac{40}{100} + \frac{40}{150} + \frac{100}{350} = 0.952380952$$

Since $U_{lub} < U < 1$ we cannot determined whether the task set is schedulable using the Processor Utilitzation Factor test.

Example 9: Processor Demand Analysis

Let us choose a set of scheduling points for the analysis.

- Task τ_1.

$$S_1 = \{100\}$$

  For each scheduling point in S_1 find, if any, point for which $W_1(t) \leq D_1$

$$W_1(0, 100) = \sum_{h=1}^{1} \left\lceil \frac{t}{T_h} \right\rceil C_h = C_1 = 40 \leq 100$$

  Since there exists at least a scheduling point where the relationship is satisfied, τ_1 is schedulable
- Task τ_2.

$$S_2 = \{100, 150\}$$

$$W_2(0, 100) = \sum_{h=1}^{2} \left\lceil \frac{t}{T_h} \right\rceil C_h = 40\frac{100}{100} + 40 \left\lceil \frac{100}{150} \right\rceil = 40 + 40 = 80 \leq 100$$

  Since there exists at least a scheduling point where the relationship is satisfied, τ_2 is schedulable
- Task τ_3.

$$S_3 = \{100, 150, 200, 300, 350\}$$

$$\begin{aligned}
W_3(0, 100) &= 40 + 40 + 100 = 180 > 100 \\
W_3(0, 150) &= 2 \cdot 40 + 40 + 100 = 220 > 150 \\
W_3(0, 200) &= 2 \cdot 40 + 2 \cdot 40 + 100 = 260 > 200 \\
W_3(0, 300) &= 3 \cdot 40 + 2 \cdot 40 + 100 = 300 \leq 300
\end{aligned}$$

  Since there exists at least a scheduling point where the relationship is satisfied, τ_3 is schedulable

Example 10: Response Time Analysis

- Task τ_1.

$$\begin{aligned}
R_1^{(0)} &= C_1 = 40 \\
R_1^{(1)} &= C_1 = R_1^{(0)}
\end{aligned}$$

  Since $R_1 < D_1$, the task is schedulable.

- Task τ_2.

$$
\begin{aligned}
R_2^{(0)} &= C_2 = 40 \\
R_2^{(1)} &= 40 + \left\lceil \frac{40}{100} \right\rceil 40 = 80 \\
R_2^{(2)} &= 40 + \left\lceil \frac{80}{100} \right\rceil 40 = R_2^{(1)}
\end{aligned}
$$

Since $R_2 \leq D_2$ the task is schedulable.

- Task τ_3.

$$
\begin{aligned}
R_3^{(0)} &= C_3 = 100 \\
R_3^{(1)} &= 100 + \left\lceil \frac{100}{100} \right\rceil 40 + \left\lceil \frac{100}{150} \right\rceil 40 = 180 \\
R_3^{(2)} &= 100 + \left\lceil \frac{180}{100} \right\rceil 40 + \left\lceil \frac{180}{150} \right\rceil 40 = 260 \\
R_3^{(3)} &= 100 + \left\lceil \frac{260}{100} \right\rceil 40 + \left\lceil \frac{260}{150} \right\rceil 40 = 300 \\
R_3^{(4)} &= 100 + \left\lceil \frac{300}{100} \right\rceil 40 + \left\lceil \frac{300}{150} \right\rceil 40 = R_3^{(3)}
\end{aligned}
$$

Since $R_3 \leq D_3$ the task is schedulable.

Real-Time Scheduling

Chapter 2

Periodic Task Scheduling

The term task is used to indicate a schedulable entity (either a process or a thread), in particular:

- A thread represents a flow of execution (it executes with shared resources, multi thread within the same process)
- A process represents a flow of execution + private resources (it executes with its own resources), such as address space, file table, ...

Tasks do not run on bare hardware, but then how can multiple tasks execute on one single CPU? The OS kernel is a piece of the operating system that takes care of multi-programming and somehow it is able to create the illusion that each CPU/processor has its own space, whereas in fact it is sharing the same resources with other processes.
In the end the kernel provides the mechanism that enable multiple tasks to execute in parallel; in a sense tasks have the illusion of executing concurrently on a dedicated CPU per task.

On this regard, with the term concurrency we refer to the simultaneous execution of multiple threads/processes in the same PC.
Concurrency is implemented by multiplexing tasks on the same CPU. Tasks are alternated on a real CPU and the task scheduler decides which task executes at a given instant in time. In other terms, in order to implement the concurrency mechanism it is necessary to introduce this new component (i.e. the task scheduler), since it makes sure that the time of your pc is shared between the different processes or tasks that compete for the reosurces at that time.

Tasks are associated to temporal constraints (a.k.a. deadlines), hence the scheduler must allocate the CPU to tasks so that their deadlines are respected.

2.1 Real Time Scheduling

Definition 18: Scheduler

A scheduler generates a schedule from a set of tasks

1. In the case of Unicore processor system (UP) (simpler definition), a schedule $\sigma(t)$ is a function mapping time t into an executing task.

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

   where $\mathcal{T}$ is the taskset and τ_{idle} is the idle task

2. For a Symmetric Multipprocessor System (SMP) (m CPUs), $\sigma(t)$ can be extended to map t in vectors $\tau \in (\mathcal{T} \cup \tau_{idle})^m$

Hence a scheduler is responsible for selecting the task to execute at time t.

Definition 19: Scheduling algorithm

Algorithm used to select for each time instant t a task to be executed on a CPU among the ready task

Given a task set $\mathcal{T}$, a scheduling algorithm $\mathcal{A}$ generates the schedule $\sigma_{\mathcal{A}}(t)$.
A task set is schedulable by an algorithm $\mathcal{A}$ if $\sigma_{\mathcal{A}}$ does not contain missed deadlines.
To verify that no missed deadlines occur, a **Schedulability test** checks if $\mathcal{T}$ is schedulable by $\mathcal{A}$.

Schedulability test

2.2 Cyclic Executive Scheduling

Timeline Scheduling, also known as **Cyclic Executive Scheduling**, is one of the most used approaches to handle periodic tasks in defense military systems and traffic control systems.
The methods consists of dividing the tmeporal axis into slots of equal length, in which one or more tasks can be allocated for execution, in such a way to respect the frequencies derived from the application requirements. A timer synchronizes the activation of the tasks at the beginning of each time slot.

Timeline Scheduling

Cyclic Executive Scheduling

Cyclic Executing Scheduling is a **static scheduling algorithm** where **jobs are not preemptable** (i.e. A scheduled job executes until termination).
The slots are statically allocated to the tasks using a **scheduling table**.

static scheduling algorithm

scheduling table

In this Scheduling algorithm two quantities are considered:

- **Major Cycle**: least common multiple of all the tasks' periods (a.k.a. **hyperperiod**)
- **Minor Cycle**: greatest common divisor of all the tasks' periods

Major Cycle

hyperperiod

Minor Cycle

The period timer fires every Minor Cycle Δ.
Hence the implementation of the scheduling algorithm performs as follow:

1. The periodic timer fires every minor cycle
2. Read the scheduling table and execute the appropriate tasks
3. Sleep until next minor cycle

The main advantage of timeline scheduling is its simplicity. The method can be implemented by programming a timer to interrupt with a period equal to the minor cycle and by writing a main program that calls the tasks in the order given in the major cycle, inserting a time synchronization point at the beginning of each minor cycle. Since the task sequence is not decided by a scheduling algorithm in the kernel, but it is triggered by the calls made by the main program, there are no context switches, so the runtime overhead is very low. Moreover, the sequence of tasks in the schedule is always the same, can be easily visualized, and it is not affected by jitter (i.e., task start times and response times are not subject to large variations).

In spite of these advantages, timeline scheduling has some problems. For example, it is very fragile during overload conditions. If a task does not terminate at the minor cycle boundary, it can either be continued or aborted. In both cases, however, the system may run into a critical situation. In fact, if the failing task is left in execution, it can cause a domino effect on the other tasks, breaking the entire schedule (timeline break). On the other hand, if the failing task is aborted while updating some shared data, the system may be left in an inconsistent state, jeopardizing the correct system behavior.
Another big problem of the timeline scheduling technique is its sensitivity to application changes. If updating a task requires an increase of its computation time or its activation frequency, the entire scheduling sequence may need to be reconstructed from scratch.
Finally, another limitation of the timeline scheduling is that it is difficult to handle aperiodic activities efficiently without changing the task sequence. The problems outlined above can be solved by using priority-based scheduling algorithms.

2.3 Fixed Priority Scheduling

Fixed Priority Scheduling is a very simple preemptive scheduling algorithm.

To each task τ_i is assigned a fixed priority p_i as an integer number: the higher the number the higher the priority. In the research literature sometimes, authoers use the opposite convention: the lowest the number, the highest the priority.

The active task with the highest priority is scheduled.

Fixed Priority Scheduling has the following priority:

- The response time of the task with the highest priority is minimum and equal to its WCET
- The reposnse time of the other tasks depends on the interference of the higher priority tasks
- The priority assignment may influence the schedulability of a task set
  Problem: how to assign tasks' priorities so that a task set is schedulable?

There are two main approaches to assigning priorities to the task set:

- **Schedulability**, i.e. find the priority assignment that makes all tasks schedulable
- **Response time (optimization)**, i.e. find the priority assignment that minimise the response time of a subset of tasks

Schedulability

Response time (optimization)

By now we consider the first objective only, hence we will investigate the **optimal priority assignment (Opt)**.

optimal priority assignment (Opt)

2.3.1 Rate Monotonic Scheduling

The Rate Monotonic (RM) scheduling algorithm is a simple rule that assigns priorities to tasks according to their request rates. Specifically, tasks with higher request rates (that is, with shorter periods) will have higher priorities. Since periods are constant, RM is a fixed-priority assignment: a priority p_i is assigned to the task before execution and does not change over time. Moreover, RM is intrinsically preemptive: the currently executing task is preempted by a newly arrived task with a shorter period.

In 1973, Liu and Lyland showed that RM is **optimal** among all fixed-priority assignments (with deadline equals to the period and offset equal to 0) in the sense that no other fixed-priority algorithms can schedule a task set that cannot be scheduled by RM.
In addition, RM is an optimal algorithm for harmonic task sets. This holds also for sporadic tasks.

2.3.2 Deadline Monotonic Scheduling

The Dealine Monotonic (DM) priority assignemt weakens the *periodi equals dealine* contraint within a static priority scheduling scheme. This algorithm was first proposed in 1982 by Leung and Whitehead as an extension of Rate Monotonic, where tasks can have relative dealines less than or equal to their period (i.e. *constrained deadlines*).

According to the DM algorithm, each task is assigned a fixed priority p_i inversely proportional to its relative deadline D_i. Thus, at any instant, the task with the shorter relative deadline is executed. Since relative deadlines are constant, DM is a static priority assignment. As RM, DM is normally used in a fully preemptive mode, that is the currently executing task is preempted by a newly arrived task with shorter relative deadline.

The DM priority assignment is **optimal**, meaning that, if a task set is schedulable by some fixed priority assignment (with deadline different from the period and offset equal to 0), then it is also schedulable by DM.
This holds also for sporadic tasks.

2.4 Dynamic Priority Scheduling

RM and DM are optimal fixed priority assignments. Maybe we can improve schedulability by using **dinamic priorities**? Assumption: priorities change from job to job (a job $J_{i,j}$ always has the same priority $p_{h,k}$)

dinamic priorities

2.4.1 Earliest Deadline First (EDF)

The Earliest Deadline First (EDF) algorithm is a dynamic scheduling rule that selects tasks according to their absolute deadlines. Specifically, tasks with realier deadlines will be executed at higher priorities. Since the absolute deadline of a periodic task depends on the current jth instance as

$$d_{i,j} = (j-1)T_i + D_i$$

EDF is a dynamic priority assignment. Moreover, it is typically executed in preemptive mode, thus the currently executing task is preempted whenever another periodic instance with realier deadline becomes active.

Note that EDF does not make any specific assumption on the periodicity of the tasks; hence, it can be used for scheduling periodc as well as aperiodic and sporadic tasks.

Chapter 3

Aperiodic Servers

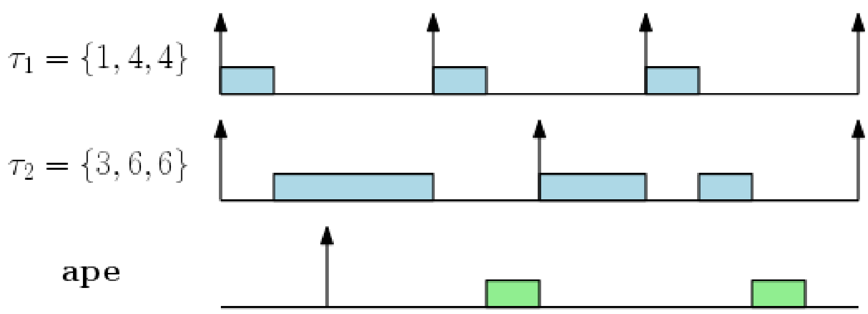
The scheduling algorithms treated in the previous chapter deals with homogeneous sets of tasks, where all computational activities are periodic. Many real-time control applications, however, require both aperiodic and periodic processes, which may also differ for their criticality. Tipically, periodic tasks are time-driven and execute critical control activities with hard timing contraints aimed at guaranteeing regular activation rates. Aperiodic tasks are usually event-driven and may have hard, soft, or non real-time requirements depending on the specific applications.

When dealing with hybrid task sets, the main objective of the kernel is to guarantee the schedulability of all critical tasks in worst-case conditions and provide good average response times for soft and non-real-time activities. Off-line guarantee of event-driven aperiodic tasks with critical timing contraints can be done only by making proper assumptions on the environment; that is, by assuming a maximum arrival rate for each critical event. This implies that aperiodic tasks associated with critical events are characterized by a minimum interarrival time between consecutive instances, which vounds the aperiodic load. Aperiodic tasks characterized by a minimum interarrival time are called sporadic. They are guaranteed under peak-load situations by assuming their maximum arrival rate.

3.1 Background Execution

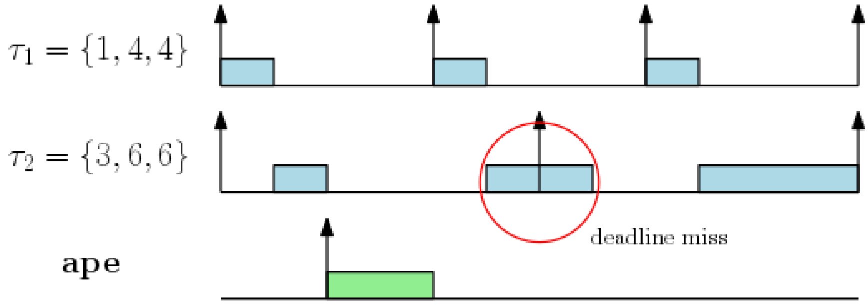
The simplest method to handle a set of soft aperiodic activities in the presence of periodic tasks is to schedule them in background; that is, when there are not periodic instances ready to execute. The major problem with this technique is that, for high periodic loads, the response time of aperiodic requests can be too long for certain applications. For this reason, background scheduling can be adopted only when the aperiodic activities do not have stringest timing constraints and the periodic load is not high.

The major advantage of background scheduling is its simplicity. In general, only two queues are needed to implement the scheduling mechanism: one (with a higher priority) dedicated to periodic tasks and the other (with a lower priority) reserved for aperiodic requests. The two queueing strategies are independent and can be realized by different algorithms. Tasks are taken from the aperiodic queue only when the periodic queue is empty. The activation of a new periodic instance causes any aperiodic tasks to be immediately preempted.



3.2 Immediate Execution

Contrary to the Background Execution, aperiodic tasks are served with the highest priority as soon as they come. This however, might cause deadline misses among the periodic tasks.



Aperiodic Servers are the solution to the problem. Normally we associate two parameters with a server:

- C_s: capacity
- T_s: server period

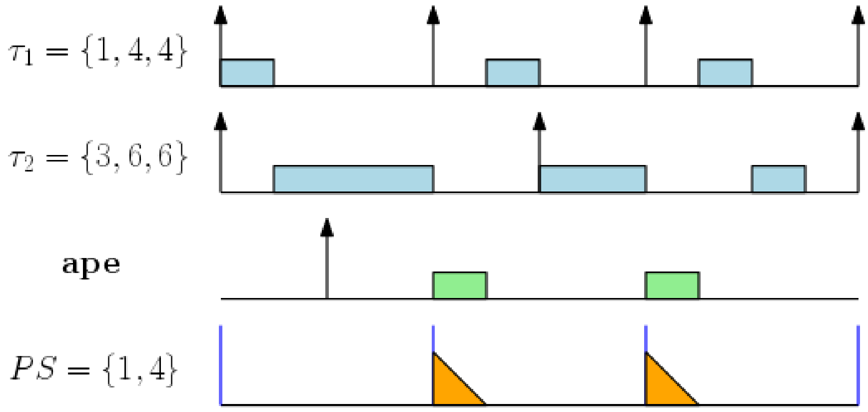
Roughly speaking, the idea is that the served tasks receive no more that C_s time units every T_s. How this is done depends on the specific server technology.

The server is scehduled as any periodic tasks. Priorities are manipulated in favour of the server. Tasks inside the server can be queued with an arbitrary discipline.

3.3 Polling Servers (PS)

The average response time of aperiodic tasks can be improved with respect to background scheduling through the use of a **server**, that is, a periodic task whose purpose is to service aperiodic requests as soon as possible. Like any periodic task, a server is characterized by a **server period** T_s and a computation time C_s, called **server capacity**, or **server budget**. In general, the server is scheduled with the same algorithm used for the periodic tasks, and once active, it serves the aperiodic requests within the limit of its budget. The ordering of aperiodic requests does not depend on the scheduling algorithm used for periodic tasks, and it can be done by arrival time, computation time, deadline or any other parameter.

server
server period
server capacity
server budget



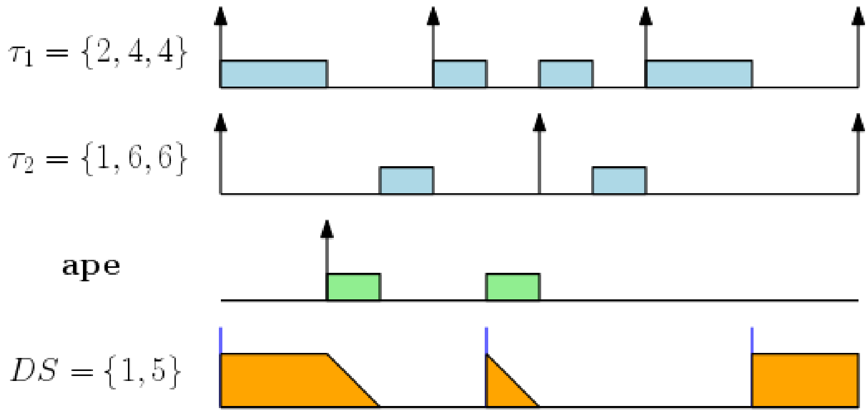
The **Polling Server (PS)** is an algorithm based on such an approach. At regular intervals equal to the period T_s, PS become active and serves the pending aperiodic requests withing the limit of its capacity C_s. If no aperiodic requests are pending, PS suspends itself until the beginning of its next period, and the budget originally allocated for aperiodic service is discharged and given periodi tasks.

Polling Server (PS)

Note that if an aperiodic request arrives just after the server has suspended, it must wait until beginning of the next period, when the server capacity is replenished at its full value.

3.4 Deferrable Servers (DS)

The **Deferrable Server (DS)** algorithm is a service technique introduced by Lehoczky, Sha, and Strosnider to improve the average response time of aperiodic requests with respect to polling service. As the Polling Server, the DS algorithm creates a periodic task (usually having a high priority) for servicing aperiodic requests. However, unlike polling, DS preserves its capacity if no requests are pending upon the invocation of the server. The capacity is maintained until the end of the period, so that aperiodic requests can be services at the same server's priority at anytime, as long as the capacity has not been exhausted. At the beginning of any server period the capacity is replenished at its full value. Deferrable Server (DS)



DS provides much better aperiodic responsiveness than polling, since it preserves the capacity until is needed. Shorter response times can be achieved by creating a Deferrable Server having the highest priority among the periodic tasks.

3.5 Sporadic Servers (SS)

The **Sporadic Server (SS)** algorithm is another technique which allow the enhancement of the average response time of aperiodic tasks without degrading the utilization bound of the periodic task set. Sporadic Server (SS)

The SS algorithm creates a high-priority task for servicing aperiodic requests and, like DS, preserves the server capacity at its high-priority level until an aperiodic request occurs. However, SS differs from DS in the way it replenishes its capacity. Whereas DS periodically replenish their capacity to full value at the beginning of each server period, SS replenishes its capacity only after it has been consumed by aperiodic task execution.

In order to simplify the description of the replenishment method used by SS, the following terms are defined:

- P_{exe} It denotes the priority level of the task that is currently executing
- P_s It denotes the priority level associated with SS
- **Active** SS is said to be active when $P_{exe} \geq P_s$
- **Idle** SS is said to be idle when $P_{exe} < P_s$
- **RT** It denotes the replenishment time at which the SS capacity will be replenished
- **RA** It denotes the replenishment amount that will be added to the capacity at time RT

Using this terminology, the capacity C_s consumed by aperiodic requests is replenished according to the following rules:

- The replenishment time RT is set as soon as SS becomes active and $C_s > 0$. Let t_a be such a time. The value of RT is set equal to T_a plus the server period

$$RT = t_a + T_s$$

- The replenishment amount RA to be done at time RT is computed when SS becomes idle or C_s has been exhausted. Let t_I be such ta time. The value of RA is set equal to the capacity consumed withing the interval $[t_a, t_I]$

3.6 Constant Bandwidth Servers (CBS)

In this section we present a novel service mechanism, called **Constant Bandwidth Server (CBS)**, which efficiently implements a bandwidth reservation strategy. The Constant Bandwidth Server guarantees that, if U_s is the fraction of processor time assigned to a server (i.e. its bandwidth), its contribution to the total utilization factor is no greater thatn U_s, even in the presence of overloads. Constant Bandwidth Server (CBS)

The basic idea behind the CBS mechanism can be explained as follows: when a new job enters the system, it is assigned a suitable scheduling deadline (to keep its demand within the reserved bandwidth) and it is inserted in the EDF ready queue. If the job tries to execute more than expected, its deadline is postponed (i.e. its priority is decreased) to reduce the interference on the other tasks. Note that by postponing the deadline, the task remains eligible for execution. In this way, the CBS behaves as a work conserving algorithm, exploiting the available slack in an efficient (deadline-based) way, thus prividing better responsiveness with respect to non-work conserving algorithms and to other reservation approaches that schedule the extra portions of jobs in background.

If a subset of tasks is handled by a single server, all the tasks in that subset will share the same bandwidth, so there is no isolation among them. Novertheless, all the other tasks in the system are protected agains overruns occurring in the subset.

In order not to miss any hard deadline, the deadline assignment rules adopted by the server must be carefully designed.

Definition 20: Constant Bandwidth Server

A CBS is characterized by three main quantities:

- an ordered pair (Q_s, T_s) assigned by the user.
  Where Q_s is the maximum budget and T_s is the period of the server. The ratio

$$U_s = \frac{Q_s}{T_s}$$

  is denotes as the server bandwidth.
- The current budget q_s (initialized to 0) managed by the server.
- The scheduling deadline d_s (initialized to 0) managed by the server.

Each served job J_k is assigned a dynamic deadline equal to the current server deadline. Whenever a served job executes, the server budget q_s is decreased by the same amount.

The CBS acts considering the following procedures:

1. When the server budget is exhausted (i.e. $q_s = 0$), the server budget is recharged at the maximum value Q_s and a new server deadline is generated as $d_s = d_s + T_s$. Note that there are no finite intervals of time in which the budget is equal to zero.

2. When a job J_k arrives and the server is active the request is enqueued in a queue of pending jobs according to a given (arbitrary) discipline.

3. When a job J_k arrives and the server is idle, if $q_s \geq (d_s - r_k)U_s$ the server generates a new deadline $d_s = r_k + T_s$ and q_s is recharged at the maximum value Q_s, otherwise the job is served with the last server deadline d_s using the current budget.

4. When a job finishes, the next pending job, if any, is served using the current budget and deadline. If there are no pending jobs, the server becomes idle.

Hence, the server behaviour can be described by the algorithm:

At arrival of job J_k at time r_k $\rightarrow$ Assign d_s
if $\exists$ pending aperiodic request **then**
  enqueue J_k
else
  if $(q_s \geq (d_s - r_k)U_s)$ **then**
    $q_s \leftarrow Q_s$
    $d_s \leftarrow r_k + T_s$
  else
    Continue to use the budget q_s with deadline d_s
  end if
end if

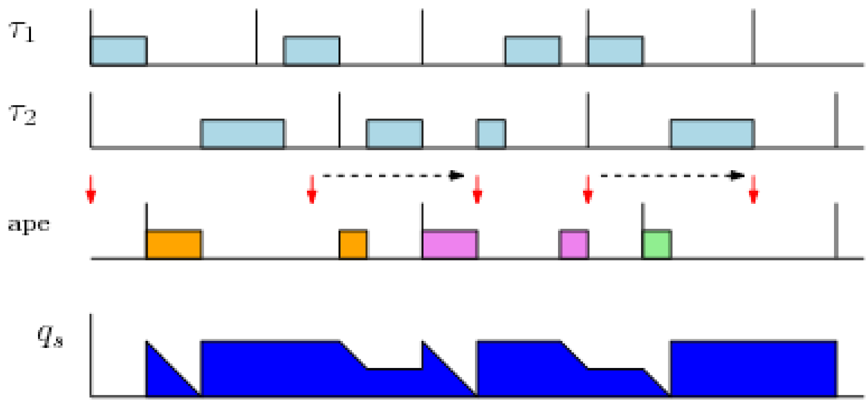


Figure 3.1: Example of two periodic task and CBS with $(Q_s, T_s) = (2, 6)$. The priority ordering is $\tau_1 > CBS > \tau_2$

3.6.1 CBS Properties

The proposed CBS service mechanism presents some interesting properties that make it suitable for supporting applications with highly variable computation times. The most important one, the **temporal isolation property**, is formally expressed as follows:

temporal isolation property

Theorem 1

The CPU utilization of a CBS with parameters (Q_s, T_s) is

$$U_s = \frac{Q_s}{T_s}$$

independently from the computation times and the arrival pattern of the served jobs.

Lemma 1

Given a set of n periodic hard tasks with processor utilization U_p and a set of m CBSs with processor utilization

$$U_s = \sum_{i=1}^{m} U_{si}$$

the whole set is schedulable by EDF if and only if

$$U_p + U_s \leq 1$$

The temporal isolation property allows us to use a bandwidth reservation strategy to allocate a fraction of the CPU time to soft tasks whose computation time cannot be easily bounded. The most imporant consequence of this result is that soft tasks can be scheduled together with hard tasks without affecting the a priori guarantee, even in the case in which the execution time of the soft

tasks are not known or the soft requests exceed the expected load.
Another general technique used in real-time systems for limiting the effects of overruns in tasks with variable computation times is the **resource reservation paradigm**. According to this method, each task is assigned a fraction of the processor bandwidth, just enough to satisfy its timing containts. The kernel, however, must prevent each task from consuming more that the requested amount to protect the other tasks in the systems (**temporal protection**). In this way, a task receiving a fraction U_i of the total processor bandwidth behaves as it were executing alone on a slower processor with a speed equal to U_i times the full speed. The advantage of this method is that each task can be guaranteed in isolation, independently of the behavior of the other tasks.

resource reservation paradigm

temporal protection

A simple and effective mechanism for implementing resource reservation in a real-time system is to reserve each task τ_i a specified amount of CPU time Q_i in every reservation period T_s.

Chapter 4

Resource Access Protocols

4.1 Introduction

A **resource** is any software structure that can be used by a process to advance its executino. Tipically, a resource can be a data structure, a set of variables, a main mamory area, a file, or a set of registers of a peripheral device. A resource dedicated to a particular process is said to be *private*, whereas a resource that can be used by more tasks is called a *shared resource*. A shared resource protexted against concurrent accesses is called an *exclusive resource*. resource

To ensure consistency of the data structures in exclusive resources, any concurrent operating system should use appropriate resource access protocols to guarantee a mutual exclusion among competing tasks. A piece of code executed under mutual exclusion contraints is called a **critical section**. critical section

Any task that needs to enter a critical section must wait until no other task is holding the resource. A task waiting for exclusive resource is said to be *blocked* on that resource, othertwise it proceeds by entering the critical section and holds the resource. When a task leaves a critical section, the resource associated with the critical section becomes *free*, and it can be allocated to another waiting task, if any.

In this chapter, we describe the main problems that may arise in a uniprocessor system when concurrent tasks use shared resources in exclusive mode, and we present some resource access protocols designed to avoid such problems and bound the maximum blocking time of each task. We then show how such blocking times can be used in the schedulability analysis to extend the guaranteee tests derived for periodic task sets.

4.1.1 Atomicity

So far we have assumed that all tasks that run and compete for a processor are independent, which means that they do not interact with one another. But there are several occasions in which this assumption cannot be made (e.g. when two tasks need to share information, exchange variables,...). The other is when you have to compete for shared resources.

In this section we will introduce this problem and in particular we will introduce the notion of atomicity.

Definition 21: Atomic Instruction

An atomic instruction is an instruction whose execution cannot be interleaved with the execution of other instructions

In this sense atomic operations are always sequentialized since they cannot be interrupted. Under these conditions they are safe operations.
On the other hand, non atomic operations can be interrupted, and as such they are not *safe* operations.

Usually, it is preferrable to have, whenever possible, non atomic operations, because they allow you to exploit in full the possibility of scheduling the processor to activities having higher priority via preemption.

Example 11: Non atomic operations

Conside a simple operation like

$$x = x + 1$$

The variable is stored in a memory address that we call x. Hence, whenever the variable is incremented using this operation, we would have to:

- load the variable x into a register $R0$

```
LD R0, x
```

- increment the register

```
INC R0
```

- store back the value contained in the register into x

```
ST x, R0
```

If the same operation is executed inside an interrupt handler an inconsistency may arise.

Example 12: Interrupt on non-atomic operations

Let us consider that the increment operation is both applied in the normal code and in an interrupt handler code (routine executed in response to an interrupt).
In both cases, the operation is translated into the assembly language using three instructions (load, increment and store).
The program starts executing as follows:

1. The normal code starts executing: the value of x is loaded from memory to the register
2. At some point during this operation something triggers the execution of the interrupt
3. The interrupt handler creates a copy of all the registers
4. The interrupt handler load (once again) x from memory, increments the register and stores the result in memory (the value of x in memory has changed to $x + 1$)
5. Upon the interrupt handler has completed, the saved registers are restored. Hence, the old value of the register (i.e. x) is loaded back, its value is incremented to $x + 1$ and stored into memory at the address of x

The problem is that even though the code should have performed two increments (i.e. the final value should have been $x + 2$), it yields the incorrect result (i.e. $x + 1$).
From a logical point of view two increment operations should have taken place, but in effect one of them not successfully completed because while i was incrementing the variable, I was allowed to be interrupted and I was left with a state that was not up to date.

The nasty problem about this phenomenon is that it does not always happen in this way, because sometimes the function successfully completes before the interrupt is fired.
The example provided is the description of a condition called **critical race**, because you can have multiple execution of your code that interleave the operation in slightly different ways and you obtain different results. `critical race`

This is a nasty problem in computer science because it might not materialize for years.

This is so because you cannot make assumption about the speed of the hardware and on the exact moment when certain events take place (we do not know the order of execution of the hardware instructions).

The case studies proposed are a perfect example of a not atomic operation that should be atomic.

The same behaviour occurs not only in the case of interrupts but also on interleaving tasks: so you could have two tasks running in parallel, both of which are sharing the variable x.

We can give a few definitions that are important for the follow up of our discussion:

Definition 22: Shared Object

An object where the conflict may happen.

Definition 23: Critical section

A critical section is a sequence of operations that cannot be interleaved with other operations on the same resource

Definition 24: Mutual exclusion

Two critical sections cannot be active at the same time (they must be sequentialized). Either one or the other needs to stand by while the other executes

There are three ways to obstain mutual exclusion:

1. Implementing the critical section as an atomic operation.
   This that interrupt are disabled before the execution of the critical section and then are restored upon termination of the operation. The problem with this is that it is really tough because disabling the interrupts the I/O system of the machine is no longer allowed to work properly. (pressing an emergency button will have no effect and the execution of the code will continue),
   Moreover if the critical section is long, no interrupt can arrive during the critical section. In the case of a timer interrupt that arrives every 1ms, if a critical section lasts more than 1ms, a timer interrupt could be lost!

2. Disabling the preemption (system-wide).
   This strategy will have some problems, because all the tasks will suffer from this suspention of the preemption even though they do not use shared resources.

3. Selectively disabling the preemption (using semaphores and mutual exclusion).
   This strategy will disable preemption only for the task that operates on the shared resource.

Hence, we should try to disable preemption rather than disabling interrupts.

Still the big issue with selectively disabling preemption the priority mechanism is no longer enforced: during the critical section might force the process to execute low priority tasks because preemption is disabled. This phenomenon is known as **Priority Inversion**. Priority Inversion

If Priority inversion is not correctly managed it may lead to a complete violation of all timing contraints.

4.1.2 Interacting Tasks

Until now, we have considered only independent tasks, which are characterized by the fact that a job never blocks or suspends and a task only blocks on job termination.
In the real world, jobs might block for various reasons:

- Tasks exchange data throught shared memory (mutual exclusion)
- A task might need to synchronize with other tasks while waiting for some data
- A job might need a hardware resource which is currently not available.

Example 13: Control Application

Let us consider a control application composed by three periodic tasks:

- τ_1 reads the data from the sensors and applies a filter. The results are stored in memory.
- τ_2 reads the filtered data and computes some control law (updating the state and the outputs); both the state and the outputs are stored in memory
- τ_3 reads the outputs and writes on an actuator

All of the three tasks access data in shared memory. This means that there are conflicts on accessing this data concurrently with the risk that the data structures become inconsistent.

4.1.3 Priority Inversion Phenomenon

The rest of this chapter presents the following resource access protocols:

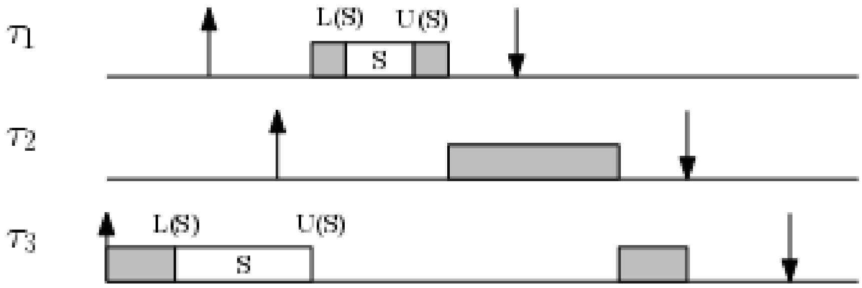
- Non-Preemptive Protocol (NPP)
- Highest Locking Priority (HLP)
- Priority Inheritance Protocol (PIP)
- Priority Ceiling Protocol (PCP)

4.2 Non Preemptive Protocol (NPP)

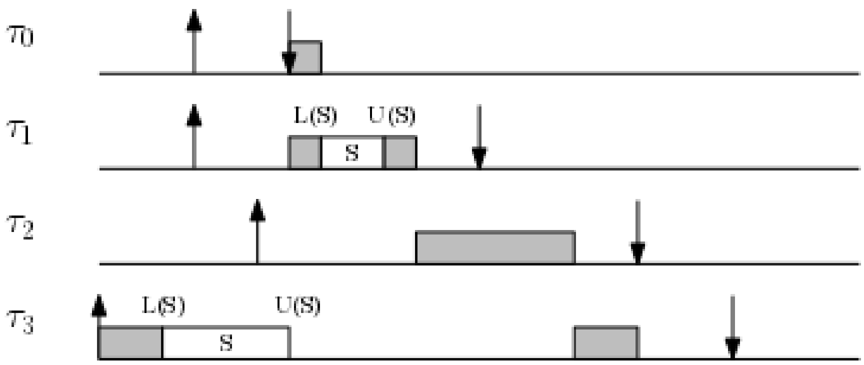
A simple solution that avoids the unbounded priority inversion problem is to disallow preemption during the execution of any critical section. This method, also referred to as **Non-Preemptive Protocol (NPP)**, can be implemented by raising the priority of a task to the highest priority level whenever it enters a shared resource. In particular, as soon as a task τ_i enters a resource R_k, its dynamic priority is raised to the level: Non-Preemptive Protocol (NPP)

$$p_i(R_k) = \max_h \{p_h\}$$

The dynamic priority is then reset to the nominal value p_i when the task exits the critical section.



This method solves the priority inversion phenomenon, however, is only appriopriate when tasks use short critical sections because it creates unnecessary blocking. This actually might cause deadline misses of tasks that do not use the shared resource.



In the example, τ_0 misses its deadline (suffers a blocking time equal to 3) even though it does not use any resource!!
The solution is to raise τ_3 priority to the maximum between tasks accessing the shared resource (i.e. τ_1) priority.

4.2.1 Blocking Time and Response Time

NPP introduces a blocking time on all tasks bounded by the maximum length of a critical section used by lower priority tasks. Such blocking time affect the response times of the task as follows:

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

where:

- B_i is the blocking time from lower priority tasks
- $\sum_{h=1}^{i-1} \left\lceil \frac{R_i}{T_h} \right\rceil C_h$ is the interference from higher priority tasks

Example 14: Response time computation

Consider the following task set:

- $\tau_1 = (20, 30, 70)$ accessing resource for $\xi_{1,1} = 0$ units of time
- $\tau_2 = (20, 45, 80)$ accessing resource for $\xi_{2,1} = 1$ units of time
- $\tau_3 = (20, 130, 200)$ accessing resource for $\xi_{3,1} = 2$ units of time

It follows that the blocking times associated with each task is the maximum length of a critical section used by lower priority tasks:

- $B_1 = 2$ since τ_3 might block the task for 2 units of time when accessing its resource
- $B_2 = 2$ since τ_3 might block the task for 2 units of time when accessing its resource
- $B_3 = 0$ since there are no lower priority tasks that can block it

Thus, the response time of each task takes the form

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

- For τ_1

$$R_1^{(0)} = 20 + 2 = 22$$

- For τ_2

$$\begin{aligned} R_2^{(0)} &= 20 + 2 = 22 \\ R_2^{(1)} &= 20 + 2 + \left\lceil \frac{22}{70} \right\rceil 20 = 42 \\ R_2^{(2)} &= 20 + 2 + \left\lceil \frac{42}{70} \right\rceil 20 = 42 \end{aligned}$$

- For τ_2

$$R_3^{(0)} = 35 + 0 = 35$$
$$R_3^{(1)} = 35 + 0 + \left\lceil \frac{35}{70} \right\rceil 20 + \left\lceil \frac{35}{80} \right\rceil 20 = 75$$
$$R_3^{(1)} = 35 + 0 + \left\lceil \frac{75}{70} \right\rceil 20 + \left\lceil \frac{75}{80} \right\rceil 20 = 95$$
$$R_3^{(1)} = 35 + 0 + \left\lceil \frac{95}{70} \right\rceil 20 + \left\lceil \frac{95}{80} \right\rceil 20 = 115$$
$$R_3^{(1)} = 35 + 0 + \left\lceil \frac{115}{70} \right\rceil 20 + \left\lceil \frac{115}{80} \right\rceil 20 = 115$$

4.3 Highest Locking Priority (HLP)

The **Highest Locking Priority (HLP)** protocol improves NPP by raising the priority of a task that enters a resource R_k to the highest priority among the tasks sharing that resource. In particular as soon as a task τ_i enters a resource R_k, its dynamic priority is raised to the level

Highest Locking Priority (HLP)

$$p_i(R_k) = \max_h \{p_i | \tau_h \text{ uses } R_k\} \tag{4.1}$$

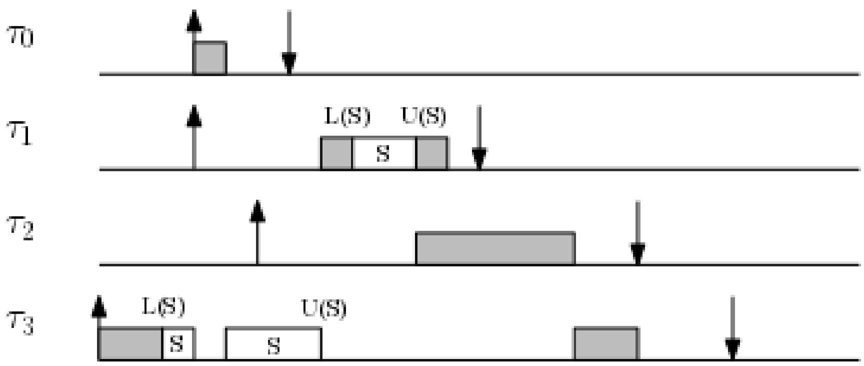
The dynamic priority is then reset to the nominal value p_i when the task exits the critical section. The online computation of the priority level in equation 4.1 can be simplified by assigning each resource R_k a **priority ceiling** $C(R_k)$ (computed offline) equal to the maximum priority of the tasks sharing R_k; that is:

priority ceiling

$$C(R_k) = \max_h \{p_i | \tau_h \text{ uses } R_k\}$$

Then, as soon as a task τ_i enters a resource R_k, its dynamic priority is raised to the ceiling of the resource. For this reason, this protocol is also referred to as **Immediate Priority Ceiling**.

Immediate Priority Ceiling



The main problem with HLP is that we must know in advance which task will access the resource: hence you need to have a complete control over the application, and how the application is written. Because knowing the code is the only way to know which resources is going to use. So this mechanism works very well if you have full control of what is running in your PC or embedded system.

4.4 Priority Inheritance Protocol (PIP)

The **Priority Inheritance Protocol (PIP)** proposed by Sha, Rajkumar and Lehoczky, avoids unbounded priority inversion by modifying the priority of those tasks that cause blocking. In

Priority Inheritance Protocol (PIP)

particular, when a task τ_i blocks one or more higher-priority tasks, it temporarily assumes (*inherits*) the highest priority of the blocked tasks. This prevents medium-priority tasks from preempting τ_i and prolonging the blocking duration experienced by the higher-priority tasks.

The Priority Inheritance Protocol can be defined as follow:

- Tasks are scheduled based on their active priorities. Tasks with the same priority are executed in a First Come First Served discipline.
- When task τ_i tries to enter a critical section and resource R_k is already held by a lower-priority task τ_j, then τ_i is blocked. τ_i is said to be blocked by the task τ_j that holds the resource. Otherwise, τ_i enters the critical section.
- When a task τ_i is blocked, it transmits its active priority to the task τ_j that holds the semaphore/mutex. Hence, τ_j resumes adn executes the rest of its critical section with a priority $p_j = p_i$. Task τ_j is said to inherit the priority of τ_i. In general, a task inherits the highest priority of the tasks it blocks. That is, at every instant,

$$p_j(R_k) = \max\{P_j, \max_h\{P_h | \tau_h \text{ is blocked on } R_k\}\} \tag{4.2}$$

- When τ_j exits a critical section, it unlocks the mutex/semaphore, and the highest-priority task blocked, if any, is awakened. Moreover, the avtive priority of τ_j is updated as follows: if no other tasks are blocked by τ_j, p_j is set to its nominal priority P_j; otherise it is set to the highest priority of the tasks blocked by τ_j, according to equation 4.2.
- Priority inheritance is transitive; that is, if a task τ_3 blocks a task τ_2, and τ_2 blocks a task τ_1, then τ_3 inherits the priority of τ_1 via τ_2

A high priority task can experience two kinds of blocking: **Direct blocking** and **Push-through blocking**.

Direct blocking

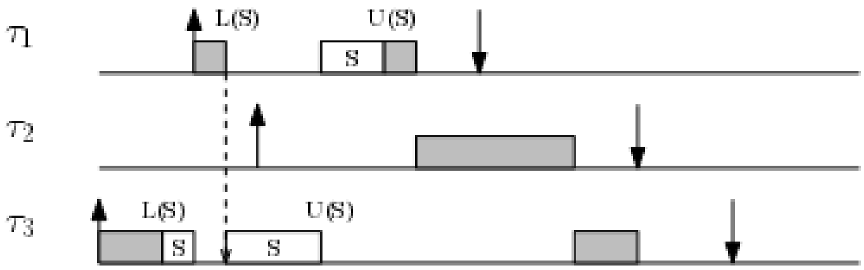
Push-through blocking

Definition 25: Direct blocking

It occurs when a higher-priority task tries to acquire a resource already held by a lower-priority task. Direct blocking is necessary to ensure the consistency of the shared resources

Definition 26: Push-through blocking

It occurs when a medium-priority task is blocked by a low-priority task that has inherited a higher priority from a task it directly blocks. Push-through blocking is necessary to avoid unbounder priority inversion



Although the Priority Inheritance Protocol bounds the priority inversion phenomenon, the blocking duration for a task can still be substantial because a chain of blocking can be formed. Another problem is that protocol does not prevent deadlocks (however, the latter problem can be solved by imposing a total ordering on the mutex accesses).

4.4.1 Blocking time and computation time

For the Priority inheritance protocol we only consider non nested critical sections. In fact, in presence of multiple inheritance, the computatino of the blocking time becomes very complex, whereas in non nested critical sections, multiple inheritance cannot happend and the computation of the blocking time becomes simpler.

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 2

If PI is used, a task block only once on each different critical section

Theorem 3

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

These two properties imply that a task can be blocker more than once, but only once per each resource and once by each task.

The Blocking time computation makes use of a **resource usage table**: resource usage table

- A task per row, in decrasing order of priority
- A resource per column
- Cell (i, j) contains $\xi_{i,j}$, i.e. the lenght of the longest critical section of task τ_i on resource S_j, or 0 if the task does not use the resource.

The computation of the blocking time makes use of the resource usage table and follows the following procedure (taking into considerations the two PI properties):

- A task can be blocked only by lower priority tasks: then, for each task (row), we must consider only the rows below (tasks with lower priority)
- A task block only on resources directly used, or used by higher priority tasks (**indirect blocking**): for each task, only consider columns on which it can be blocked (used by itself or by higher priority tasks) indirect blocking

Let us consider the following resource usage table:

	S_1	S_2	S_3
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

Example 15

- B_1
  τ_1 can be blocked only on S_1. Therefore, we must consider only the first column, and take the maximu, which is 3.
  Therefore $B_1 = 3$.
- B_2
  τ_2 can be blocked on S_1 (indirect blocking) by lower priority tasks inheriting a higher priority and on S_2 by lower priority tasks.
  Consider all cases where two distinct lower priority tasks in $\{\tau_3, \tau_4, \tau_5\}$ access S_1 and S_2, sum the two contributions, and take the maximum:

  - τ_4 on S_1 and τ_5 on S_2: blocking time of 5
  - τ_4 on S_2 and τ_5 on S_1: blocking time of 4

  Hence, $B_2 = 5$
- B_3

τ_3 can be blocked on all 3 resources

- τ_4 on S_1 and τ_5 on S_2: blocking time of 5
- τ_4 on S_1 and τ_5 on S_3: blocking time of 4
- τ_4 on S_2 and τ_5 on S_1 or S_3: blocking time of 4
- τ_4 on S_3 and τ_5 on S_2: blocking time of 3
- τ_4 on S_3 and τ_5 on S_1: blocking time of 2

Hence, $B_3 = 5$

- B_4
  τ_4 can be blocked on all 3 resources, since it can be blocked only by τ_5
  - τ_5 on S_2: blocking time of 2
  - τ_5 on S_1 or S_3: blocking time of 1

  Hence, $B_4 = 2$
- τ_5 cannot be blocked by any other task (because it is the lower priority task).
  Hence, $B_5 = 0$

As usual the schedulability tests with a blocking time can be carried on using the three tests that we have seen:

- Processor utilization factor test.
  The system is schedulable if

$$\forall i \in [1, n] \qquad \sum_{k=1}^{i-1} \frac{C_k}{T_k} + \frac{C_i + B_i}{T_i} \leq i(2^{1/i} - 1)$$

- Response time analysis

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

- Processor demand analysis.
  In a task set $\mathcal{T}$ composed of independent and periodic tasks, τ_i is schedulable (for all possible phasing) if and only if

$$\exists t \in [0, D_i] \qquad W_i(0, t) = C_i + \sum_{h=1}^{i-1} \left\lceil \frac{t}{T_h} \right\rceil C_h \leq t - B_i$$

  As usual we can define

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

$$L_i(t) = \frac{W_i(t)}{t}$$

$$L_i = \min_{t \in [0, D_i]} L_i(t) + \frac{B_i}{t}$$

  The task set is schedulable if $\forall i$, $L_i \leq 1$.
  Again, we can compute L_i by only considering the scheduling points.

4.5 Priority Ceiling Protocol (PCP)

The **Priority Ceiling Protocol (PCP)** was introduced by Sha, Rajkumar, and Lehoczky to bound the priority inversion phenomenon and prevent the formation of deadlocks and chained blocking. Priority Ceiling Protocol (PCP)

The basic idead of this method is to extend the Priority Inheritance Protocol with a rule granting a lock request on a free mutex. To avoid multiple blocking, this rule does not allow a task to enter a critical seciton if there are locked mutexes that could block it. This means that once a task enters its first critical section, it can never be blocked by lower-priority tasks until its completion.

In order to realize this idea, each mutex is assigned a priority ceiling equal to the highest priority of the tasks that can lock it. Then, a task τ_i is allowed to enter a critical section only if its priority is higher than all priority ceiling of the mutexes currently locked by tasks other than τ_i.

4.5.1 Original Priority Ceiling Protocol (OPCP)

The **Original Priority Ceiling Protocol** can be defined as follows:

Original Priority Ceiling Protocol

4.5.2 Immediate Priority Ceiling Protocol (IPCP)

Operating System Structure

Chapter 5

The Kernel

Recall the following elementary defintions:

Definition 27: Real-Time Operating Sytems (RTOS)

Operating System providing support to Real-Time applications

Definition 28: Rea-Time application

The correctness depends not only on the output values, but also on the time when such values are produced

Definition 29: Operating System

An Operating System is:

- Set of computer programs
- Interface between applications and hardware
- a way to control the execution of application programs
- a way to manage the hardware and software resources

In particular we can interpret an operating system as a:

- a Service Provider: an API plus some services behind. This API needs to be suitable for real-time application.
- a Resource Manager: implements schedulers, policies to share resource, aperiodic servers, ...

The services provided by the Operating System are executed in **Kernel Space**: whenever you execute a program on the operating systems the processor switches in a particular mode called **supervisor mode** (in this mode the processor can do anything allowed by the machine including interrupts, manage memory, ...). In this kernel space the operating system is able to: `Kernel Space` `supervisor mode`

- Process Synchronization, Inter-Process Communication
- Schedule processes/threads
- Input and Output
- Allocate Virtual Memory

All of these things are exported and accessible by means of an API.

5.1 Introduction

The core of the machine is called the **Kernel** `Kernel`

Definition 30: Kernel

core part of the OS, allowing multiple tasks to run on the same CPU

The core feature of the kernel is that it allows a task set $\mathcal{T}$ composed by N tasks to run in parallel in a M CPUs ($M < N$). From the application/program point of view there is no difference between having a task executed in an intermediate way or having tasks executed in a dedicated processor: so what the operating system does is to ensure a proper temporal multiplexing between the tasks.

The task scheduling service relies on two core components:

- **Scheduler**: decides which task to execute
- **Dispatcher**: component that implements the context switch between the tasks. Often this component relies on some hardware extension that facilitates this type of service.

Scheduler

Dispatcher

The kernel also provides a mechanism for allowing tasks to communicate and synchronize between each other. On this regard there are two possible paradigms:

- Shared memory (threads).
  Shared Memory utilizes mutexes, semaphores and condition variables, which the kernel provides. From a real time point of view this service has to come along some real-time resource sharing protocols.
- Message passing (processes).
  Contrary to Shared memory, Message passing is based on different interaction models such as pipeline, client-server, . . .
  The kernel must once again provide some IPC mechanism: pipes, message queues, mailboxes, remote procedure calls (RPC), . . .
  On top of this some real-time protocols can still be used

In practice, an adequate scheduling of system resources removes the need for over-enginnering the system, and is necessary for providing a predictable QoS. For this reason, an adequate scheduling of system resources considers two important aspects: the algorithm and the implementation.
For instance, we have seen efficient algorithm for scheduling tasks and resources, however all applications are treated by a **Timer**. This introduces a set of questions and problems to consider in the implementation of the scheduling algorithm:

Timer

- Is the timer reliable?
- Is the scheuduler able to select a high-priority task as soon as it is ready?
- And the dispatcher?

Example 16: Periodic Task

When considering a periodic task, it expects to be executed at time $r = r_0 + jT$, but sometimes it is delayed to $r = r_0 + jT + \delta$, where the quantity δ becomes an offset or a drift term that makes the timing not reliable.
This delay may cause deadline misses.

5.2 Kernel Latency

Definition 31: Kernel Latency

Delay δ with which a kernel implement its decisions

When a primitive is called, the primitive takes some time to execute and during this time there is not possibility preempt the kernel. Therefore, in practice, the operating system is temporarily sospending the scheduling mechanism. This is situation really similar to the resource access protocols,

in fact, we can think of the kernel as a shared resource that is not preemptable and therefore the kernel latency can be modelled as a blocking time.
Hence the schedulability analysis tools introduced are modified as follows:

- **Processor Utilization factor test**

$$\forall i \in [1, n] \qquad \sum_{k=1}^{i-1} \frac{C_k}{T_k} + \frac{C_i + \delta}{T_i} \leq U_{lub}$$

- **Response Time Analysis**

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

- **Processor Demand analysis**

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

The scheduler is called whenever an internal (e.g. IPC, signal, ...) or external (e.g. interrupt) events is triggered.
When one of these events take place, there exists some time between the triggering of the event and the dispatch which can be decomposed in four components:

- Event generation
- Event delivery (interrupts may be disabled)
- Scheduler activation (non preemptable section)
- Scheduling time

There are quite a few elements that may generate the delay.
All these elements of uncertainty need to be understood, the reason and how the system manages introduces this delay and most importantly how to manage this delay in a real-time kernel.
Remember that we are interested in the worst case possibility, so all the components that compose the kernel latency should be designed so that in the worst-case the maximum is minimized.

5.3 System Architecture

The system architecture is composed by a system bus that is interconnecting the following components:

- One or more CPUs
- Memory (RAM)
- I/O Devices
  - Secondary memory (disks, ...)
  - Network cards
  - Graphic cards
  - Keyboard, mouse, ...

5.3.1 The CPU

The model of the CPU is composed of the following registers

- General-purpose registers that can be accessed by all the programs. These registers can be either data registers or address registers
- Program Counter (PC) aka Intruction pointer
- Stack Pointer (SP) register
- Flags register (aka Program Status Word)
- Some special registers, which control how the CPU works, must be "protected"

Regual user programs should not be allowed to influence the CPU mode of operation, perform I/O operations and reconfigure virtual memory. For this reason there is a need for privileged mode of execution

- Regual registers vs special registers
- Regual intructions vs privileged instructions

User programs: low privilge level (**User Level**) User Level

The OS kernel runs in supervisor mode.

Example 17: Intel x86

Real CPUs are more complex. They have few General Purpose registers: EAX, EBX, ECX, EDX (accumulator registers containing an 8 bit part and a 16 bit part), EBP, ESI, EDI

- EAX: Main accumulator
- EBX: sometimes used as base for arrays
- ECX: sometimes used as counter
- EBP: stack base pointer (for subroutines calls)
- ESI: source index
- EDI: destination index

They also have segmented memory architecture: segment registers CS (code segment), DS (data segment), SS (stack segment), GS, FS.
Finally they have various mode of operation: RM, PM, VM86, x86-64,. . . , mainly due to backward compatibility.

The Kernel is part of the OS which manages the hardware.
Runs with the CPU in Supervisor Mode (high privilege level):

- Privilege level known as Kernel Level (KL), execution in Kernel Space
- Regual programs run in User Space

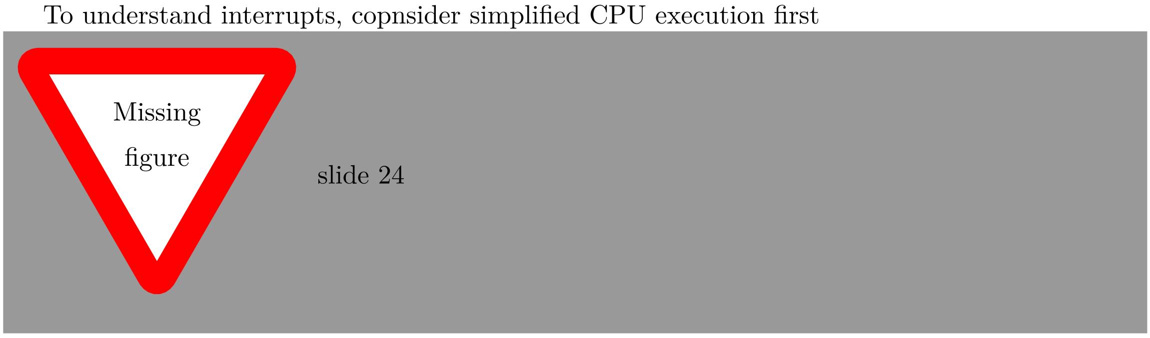
Mechanismns for increasing the privilege level (from US to KS) in a controlled way

- Interrupts (+ traps/hw exeptions).

- Instructions causing a hardware exception

Switch the CPU from User Level to Supervisor mode by entering the kernel. This can be used to implement system calls. A partical Context Switch is performed: flags and PC are pushed to the stack, if the processor is executing at User Level, switch to Kernel Level, and eventually switch to a kernel stack finally execution jumps to a handler in the kernel (save the user registers for restoring them later). Once finished return to low privilege level (execution returns to User Space) through a "return from interrupt" Assembly instruction (`IRET` on x86): Pop flags and PC from stack and eventually switch back to user stack. Return path from system calls and hardware interrupts to handlers.

To understand interrupts, copnsider simplified CPU execution first



The CPU interatively:

- Fetch an instruction (address given by PC)
- Increase the PC
- Execute the intruction (might update the PC on jump...)

A More realistic execution model

Interrupt cannot fire during the executrion of an intruction. Hardware exception: cause by the execution of an intruction:

- `trap`,`syscall`,`sc`,...
- I/O instructions at low privilege level, Page faults, ...

The interrupt tabole holds the addresses of the handlers:

- Interrupt n fires: after eventually switching to KS and pushing flags and PC on the stack
- Read the address contained in the n^{th} entry of the interrupt table, and jump to it!

Interrupt tables are implemented in hardware or in software:

- x86, interrupt description table composed by interrupt gates. The CPU automatically jumpts to the n^{th} interrupt gate
- Other CPUs jump to a fixed address: a software demultiplexer reads the interrupt table

Software Interrupt - System Call

1. Task τ_1, executes and invokes a system call
2. Execution passes from US to KS (change stack, push PC and flags, increase privilege level)
3. The invoked syscall executes. Maybe, it is blocking
4. τ_1 blocks and the system returns to US, and τ_2 is scheduled

Hardware Interrupt

1. Task τ_2 is executing, a hardware interrupt fires
2. Execution passes from US to KS (change stack, push PC and flags, increase privilege level)
3. the proper Interrupt Service Routine executes
4. The ISR can unblock τ_1. When execution returns to US, τ_1 is scheduled

The executino flow enters the kernel for two reasons

- Reacting to events coming from up (syscalls)
- Reacting to an event coming from below (an hardware interrupt from a device)

The kernel executes in the context of the interrupted task.
A system call can block the invoking task, or can unblock a different task
An ISR can unblock a task
If a task is blocked/unblocked, when returning to user space a context switch can happen.

The scheduler is invoked when return from KS to US

Example 18: I/O operation

Consider a generic Input or Output to an external device such as a PCI card.
This operation is performed by the kernel and user programs must uyse a syscall
The operation is performed in 3 phases:

1. Setup: prepare the device for the I/O operation
2. Wait: wait for the end of the operation
3. Cleanup: complete the operation

This can be done using polling, PIO, DMA, ...

5.3.2 Polling

User programs invoke the kernel; execution in kernel space until the operation is terminated.
The kernel cyclically reads (polls) an interafce status register to check if the operation is terminated
Busy-waiting in kernel space!

- No user task can execute while waiting for the I/O operation ...
- The operation must be very short
- I/O operation == blocking time

1. The user program raises a software input
2. Setup phase - in kernel: in case of input operation, nothing is done; in case of output operation, write a value to a card register
3. Wait - in kernel: cycle until a bit of the card status register becomes 1
4. Cleanup - in kernel: in case of input, read a value from a card register; in case of output, nothing is done. Eventually return to phase 1
5. IRET

5.3.3 Programmed I/O

User programs invoke tyhe kernel; execution returns to user space while waiting for the device: the task that invoked the syscall blocks
Anb interrupt will notify the kernel when the "wait" phase is terminated:

- The interrupt handler will take care of performing the I/O operation
- Many frequent short interruption of unrelated user-space tasks

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 I/O
3. Wait - return to user space: block the invoking task, and schedule a new one (IRET)
4. Cleanup - in kernel: the interrupt fires → enter kernel, and perform the I/O operation
5. Return to phase 2, or unblock the task if the operation is terminated (IRET)

5.3.4 DMA

User programs invoke the kernel; execution returns to user space while waiting for the device. The task that invoked the syscall blocks!

I/O operations are not performed by the kernel on interrupt, Performed by a dedicated HW device,
An interrupt is raised when the whole I/O operation is terminated

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
3. Wait - return to user space: block the invoking task, and schedule a new one (IRET)
4. Cleanup - in kernel: the interrupt fires $\rightarrow$ the operation is terminated. Stop device and DMA
5. Unblock the task and invoke the scheduler (IRET)

Chapter 6

Timer and Clock Latency

Definition 32: Latency

Measure of the diffence between the theoretical and actual schedule.

Example 19

A task τ expects to be scheduled at time t, but is actually scheduled at time t'. The resulting latency L takes the form:

$$L = t' - t$$

The latency L can be modelled as a blocking time and as such affects the guarantee test. Similar to what done for shared resources.
Blocking time due to latency, not to priority inversion.

Upper bound for L? if not known, no schedulability tests!! The latency must be bounded:

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

If L^{max} is too high, only few task sets result ot be schedulable.
Large blocking time experienced by all tasks!
The worst-case latency L^{max} cannot be too high.

A task τ_i is a stream of jobs $J_{i,j}$ arriving at time $r_{i,j}$.
Job $J_{i,j}$ is schedulable at time $t' > r_{i,j}$
$t' - r_{i,j}$ is given by:

1. $J_{i,j}$'s arrival is signalled at time $r_{i,j} + L^1$
2. Such event is served at time $r_{i,j} + L^1 + L^2$
3. $J_{i,j}$ is actually scheduled at $r_{i,j} + L^1 + L^2 + L^3$

where:

- L^1 is due to the delayed interrupt generation.
  Hardware interrupts; generated by devices.
  Sometimes, an interrupt should be generated at time t but it is actually generated at time $t' = t + L^{int}$ where L^{int} iis the Interrupt Generation Latency. Such latency is due to hardware issues and it is generally small compared to L^{np}. The only exception is if the device is a timer device, the interrupt generation latency can be quite high Timer resolution latency L^{timer}
- L^2 is the non-preemptable section latency (L^{np}).
  Delay between time when an event is generated and when the kernel handles it.
  Due to non-preemptable sections in the kernel, which delay the response to hardware interrupts.
  It is composed by various parts: interrupt disabling, bottom halves dealying,...
  It depends on how the kernel hadles the various events,...

- L^3 is the scheduler latency. Which is the interference from higher priority tasks and its already accounted by the guarantee tests. Hence it will not be considered

The Timer Resolution Latency is the interrupt generation latency for a hardware timer device. L^{timer} can often be much larger that the non-preemptable section latency L^{np}.
Where does it come from? Kernel timers are generally implemented by using a hardware device that produces periodic interrupts.
Can we do anything about it?
A Periodic timer interrupt is called a tick
Example: periodic task (`setitimer()`, Posix timers, `clock_nanosleep()`,...) τ_i with period T_i
Job end $\rightarrow$ τ_i sleeps for the next activation.
Activations are triggered by the periodic interrupt:

- Periodic tick interrupt, with period T^{tick}
- Every T^{tick}, the kernel checks if the task must be woken up
- If T_i is not multiple of T^{tick}, τ_i experiences a timer resolution latency

Traditional operating systems: timer device programmed to generate a periodic interrupt.

Example 20

In a PC, the Programmable Interval Timer (PIT) is programmed in periodic mode

At every tick the execution enter kernel space.
The kernel executes and can:

- Wake up tasks
- Adjust tasks priorities
- Run the scheduler, when returning to user space (possible preemption)

Timer interrupt period: trade-off between responsiveness (low latency) and throughput (low overhead).

- Large T^{tick}: large timer resolution latency
- Small T^{tick}: high number of interrupts.
  More switches between US and KS, tasks are interrupted more often and a resulting large overhead

For non real-time systems, it is possible to find a reasonable tradeoff but it still depends on the workload.

Example 21: Linux Kernel

- Linux 2.4: 10 ms (100Hz)
- Linux 2.6: 100 Hz, 250 Hz or 1000Hz
- Other systems: $T^{tick} = 1/1024$

The timer resoultion latency is experienced by all tasks that want to sleep for a specified time T.
τ_i must wake up at time $r_{i,j} = jT_i$, but is woken up at time $t' = \left\lceil \frac{r_{i,j}}{T^{tick}} \right\rceil T^{tick}$.

The Timer Resolution Latency is bounded:

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

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

Reducing T^{tick} below 1ms is generally not acceptable, so periodic tasks can expect a blocking time due to L^{timer} up to 1ms. How loarge is the effect on the schedulability tests?
Additional problems:

- Tasks' periods are rounded to multiples of T^{tick}
- Limit on the minimum task period: $\forall i, T_i \geq T^{tick}$
- A lot of useless timer interrupts might be generated

Remember?

Definition 33: Timer

generate an event at a specified time t

Definition 34: Clock

keep track of the current system time

A timer can be used to wake up a periodic task τ, a clock can be used to read the system time (`gettimeofday()`)

Definition 35: Timer Resolution

minimum interval at which a periodic timer can fire. If periodic ticks are used, the timer resolution is T^{tick}

Definition 36: Clock Resolution

minimum difference between two different timer returned by the clock.

What's the expected clock resolution?

- Traditional OSs use a "tick counter".
  Very fast clock: return the number of ticks (jiffies in Linux) from the system boot
  Clock resolution : T^{tick}
- Modern PCs have higher resolution time sources...
  On x86, TSC (TimeStamp Counter)
  High-resolution clock: use the TSC to compute the time since the last timer tick...
- In summary: high-resolution clocks are easy: every modern OS kernel provide them.
- Even using a "traditional" periodic timer tick, it is easy to provide high-resolution clocks: time can be easily read with a high accuracy.
- On the other hand, timer resolution is limited by the system tick T^{tick}. It is impossible to generate events at arbitrary instants in time, without latencies

6.1 Timer Devices

Timer Devices (e.g. PIT - i8254) generally work in 2 modes: periodic and one-shot.
Programmed writing a value C in a counter register.
The counter register is decremented at a fixed rate.
When the counter is 0, an interrupt is generated:

- If the device is programmed in periodic mode, the counter register is automatically reset to the programmed value
- If the device is programmed in one-shot mode, the kernel has to explicitly reprogram the device (setting the counter register to a new value)

The periodic mode is easier to use! This is why most kernels use it.
When using one-shot mode, the timer interrupt handler must:

1. Acknowledge the interrupt handlerm, as usual

2. Check if a timer expired, and do its usual stuff...

3. Compute when the next timer must fire

4. Reprogram the timer device to generate an interrupt at the correct time

Steps 3 and 4 are particularly critical and difficult.
When the kernel reprograms the timer device (step 4), it must know the current time, but the last known time is the time when the interrupt fired (before step 1):

- A timer interrupt fires at time t_1
- The interrupt handler starts (enter KS) at time t'_1
- Before returning to US, the timer must be reprogrammed, at time t''_1
- Next interrupt must fire at time t_2; the counter register is loaded with $t_2 - t_1$
- Next interrupt will fire at $t_2 + (t''_1 - t_1)$

The error described previously accumulates with the risk of drift between real time and system time.
A free run counter (not stopped at t_1) is needed.
The counter is synchronised with the timer device and the value of the counter at time t_1 is known.
This permits to know the time t''_1. The new counter register value can be computed correctly.
On a PC, the second PIT counter, or the TSC, or the APIC timer can be used as a free run counter.

Serious real-time kernels use high-resolution timers (use hardware time in one-shot mode) which for instance is already implemented in RT-Mach, RTLinux, RTAI and others.
General purpose kernels are more concerned about stability and overhead.

Compatibility with "traditional" kernels:

- The tick event can be emulated through high-resolution timers
- Timer device programmed to generate interrupts both: when needed to serve a timer and at tick boundaries but the "tick" concept is now useless (e.g. Tickless or NO_HZ system which are good for saving power)

Chapter 7

The Non Preemptable Section Latency

Definition 37: Non-Preemptable Section Latency

Delay between time when an event is generated and when the kernel handles it.

- Due to non-preemptable sections in the kernel, which delay the response to hardware interrupts
- Composed by various parts: interrupt disabling, bottom halves delaying,...
- Depends on how the kernel handles the various events

The non-preemptable section latency L^{np} is given by the sum of different components:

1. Interrupt disabling
2. Delayed interrupt service
3. Delayed scheduler invocation

The first two are mechanisms used by the kernel to guarantee the consistency of internal structures. The third mechanism is sometimes used to reduce the number of preemptions and increase the system throughput.

7.1 Interrupt disabling

Before checking ifg an interrupt is fired, the CPU checks if interrupts are enabled. Every CPU has some protected intructions (`STI/CLI` on x86) for enabling/disabling interrupts.
In modern system, only the kernel (or code running in KS) can enable/disable interrupts.
Interrupts disabled for a time $T^{cli} \rightarrow L^{np} \geq T^{cli}$
Interrupt disabling is used to enforce mutual exclusion between sections of the kernel and ISRs.

7.2 Dealyed Interrupt Service

When the interrupt fires, the ISR is ran, but the kernel can delay the interrupt service some more...

- ISRs are generally small, and do only few things
- An ISR can set some kind of software flag, to notify that the interrupt fired
- Later, the kernel can check such flag and run a larger (and more complex) interrupt handler

Hard IRQ handlers (ISRs) vs "Soft IRQ handlers".

The advantages of "soft IRQ handlers" are:

- ISRs generally run with interrupts disabled
- Soft IRQ handlers can re-enable hardware interrupts
- Enabling/Disabling soft handlers is simpler/cheaper

Disadvantages:

- Increase NP latency: $L^{np} >> T^{cli}$
- Soft IRQ handlers are often non-preemptable increasing the latency for other tasks too...

7.3 Delayed scheduler invocation

Scheduler invoked when returing from KS to US. Sometimes, return to US after a lot of activities

- Try to reduce the numbert of KS - US switches
- Reduce the number of context switches
- Throughput vs low latency

ISR executed at the correct time, soft IRQ handler ran immediately, but scheduler invoked too late

7.4 Summary

L^{np} depends on some different factors.
In general, no hardware reasons, it almost entirely depends on the kernel structure: non-preemptable section latency is generally the result of the strategy used by the kernel for ensuring mutual exclusion on its internal data structures.

To analyze/reduce L^{np}, we need to understand such strategies.
Different kernels, based on different structures, work in different ways.
Some activities causing L^{np} are: interrupt handling (device drivers) and management of the parallelism

Additional information and proofs

Appendix A

U_{lub} for RM for N tasks

Given a task set $\mathcal{T}$ of N tasks scheduled using the RM priority assignment, the conditions that allow to compute the least upper bound of the processor utilization factor are:

$$\begin{cases} T_1 < T_n < 2T_1 \\ C_1 = T_2 - T_1 \\ C_2 = T_3 - T_2 \\ \dots \\ C_{n-1} = T_n - T_{n-1} \\ C_n = T_1 - \displaystyle\sum_{i=1}^{n-1} C_i = 2T_1 - T_n \end{cases}$$

Thus the processor utilization factor becomes

$$T = \frac{T_2 - T_1}{T_1} + \frac{T_3 - T_2}{T_2} + \dots + \frac{T_n - T_{n-1}}{T_{n-1}} + \frac{2T_1 - T_n}{T_n}$$

Defining

$$R_i = \frac{T_{i+1}}{T_i}$$

and noting that

$$\prod_{i=1}^{n-1} R_i = \frac{T_n}{T_1}$$

the utilization factor may be written as

$$U = \sum_{i=1}^{n-1} R_i + \frac{2}{\prod_{i=1}^{n-1} R_i} - n$$

To minimize U over R_i, $i = 1, \dots, n-1$, we have

$$\frac{\partial U}{\partial R_k} = 1 - \frac{2}{R_i \prod_{i=1}^{n-1} R_i}$$

Thus defining $P = \prod_{i=1}^{n-1} R_i$, U is minimum when:

$$\begin{cases} R_1 P = 2 \\ R_2 P = 2 \\ \dots \\ R_{n-1} P = 2 \end{cases}$$

that is, when all R_i have the same value

$$R_1 = R_2 = \cdots = R_{n-1} = 2^{1/n}$$

Substituting this value in U we obtain

$$\begin{aligned} U_{lub} &= (n-1)2^{1/n} + \frac{2}{2^{(1-1/n)}} - n \\ &= n2^{1/n} - 2^{1/n} + 2^{1/n} - n \\ &= n(2^{1/n} - 1) \end{aligned}$$

Appendix B

U_{lub} for RM + Polling Server

We first consider the problem of guaranteeing a set of hard periodic tasks in the presence of soft aperiodic tasks handled by a Polling Server. Then we show how to derive a schedulability test for hard aperiodic requests.

The schedulability of periodic tasks can be guaranteed by evaluating the interference introduced by the Polling Server on periodic execution. In the worst case, such an interference is the same as the one introduced by an equivalent periodic task having a periodi equal to T_s and a computation time equal to C_s. In fact, independently of the number of aperiodic tasks handled by the server, a maximum time equal to C_s is dedicated to aperiodic requests at each server period. As a consequence, the processor utilization factor of the Polling Server is

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

and hence the schedulability of a periodic set with n tasks and utilization U_p can be guaranteed if

$$U_p + U_s \leq U_{lub}(n+1)$$

If periodic tasks (including the server) are scheduled by RM, the schedulability test becomes

$$\sum_{i=1}^{n} \left(\frac{C_i}{T_i} \right) + \frac{C_s}{T_s} \, le(2^{1/(n+1)} - 1)(n+1)$$

Note that more Polling Servers can be created and execute concurrently on different aperiodic task sets.
In general, in the presence of m servers, a set of n periodic tasks is schedulable by RM if

$$U_p + \sum_{j=1}^{m} U_s j \leq U_{lub}(n+m)$$

A more precise schedulability test can be derived by assuming that PS is the highest-priooority task in the system. To simplify the computation, the worst-case relations among the tasks are first determined, and then the lower bound is computed against the worst-case model.

Consider a set of n periodic tasks $(\tau_1, \ldots, \tau_n)$ ordered by increasing periods, and a PS server with highest priority. The worst-case scenario for a set of periodic tasks that fully utilize the processor is characterized by the following parameters:

$$\begin{cases} C_s = T_1 - T_s \\ C_1 = T_2 - T_1 \\ C_2 = T_3 - T_2 \\ \ldots \\ C_{n-1} = T_n - T_{n-1} \\ C_n = T_s - C_s - \sum_{i=1}^{n-1} C_i = 2T_s - T_n \end{cases}$$

The resulting utilization is then

$$\begin{aligned}
U &= \frac{C_s}{T_s} + \frac{C_1}{T_1} + \cdots + \frac{C_n}{T_n} \\
&= U_s + \frac{T_2 - T_1}{T_1} + \cdots + \frac{T_n - T_{n-1}}{T_{n-1}} + \frac{2T_s - T_n}{T_n} \\
&= U_s + \frac{T_2}{T_1} + \cdots + \frac{T_n}{T_{n-1}} + \left(\frac{2T_s}{T_1}\right) \frac{T_1}{T_n} - n
\end{aligned}$$

Defining:

$$\begin{cases}
R_s = \dfrac{T_1}{T_s} \\
R_i = \dfrac{T_{i+1}}{T_i} \\
K = \dfrac{2T_s}{T_1} = \dfrac{2}{R_s}
\end{cases}$$

and noting that

$$R_1 R_2 \ldots R_{n-1} = \prod_{j=1}^{n-1} R_j = \frac{T_n}{T_1}$$

The utilization factor may be written as

$$U = U_s + \sum_{i=1}^{n-1} R_i + \frac{K}{\prod_{j=1}^{n-1} R_j} - n$$

we minimize U over R_i, $i = 1, \ldots, n-1$. Hence,

$$\begin{aligned}
\frac{\partial U}{\partial R_i} &= \cancelto{0}{\frac{\partial U_s}{\partial R_i}} - \cancelto{0}{\frac{\partial n}{\partial R_i}} + \cancelto{0}{\frac{\partial \sum_{j \neq i}^{n-1} R_j}{\partial R_i}} + \frac{\partial R_i}{\partial R_i} + \frac{\partial K \left(\prod_{j=1}^{n-1} R_j\right)^{-1}}{\partial R_i} \\
&= 1 + K \left(\prod_{j=i}^{n-1} R_j\right)^{-1} \frac{\partial R_i^{-1}}{\partial R_i} \\
&= 1 - K \left(\prod_{j=i}^{n-1} R_j\right)^{-1} R_i^{-2} \\
&= 1 - \frac{K}{R_i^2 \left(\prod_{j \neq i}^{n-1} R_j\right)} \\
&= 1 - \frac{K}{R_i \left(\prod_{j=1}^{n-1} R_j\right)}
\end{aligned}$$

Thus, defining $P = \prod_{j=1}^{n-1} R_j$, U is minimum when:

$$\begin{cases}
R_1 P = K \\
R_2 P = K \\
\ldots \\
R_{n-1} P = K
\end{cases}$$

that is, when all R_i have the same value:

$$R_1 = R_2 = \cdots = R_{n-1} = K^{1/n}$$

Substituting this value in U we obstain:

$$
\begin{aligned}
U_{lub} &= U_s + \sum_{i=1}^{n-1} R_i + \frac{K}{\prod_{j=1}^{n-1} R_j} - n \\
&= U_s + \sum_{i=1}^{n-1} \left(K^{1/n}\right) + \frac{K}{\prod_{j=1}^{n-1} \left(K^{1/n}\right)} - n \\
&= U_s + (n-1)K^{1/n} + \frac{K}{K^{n\,-\,1/n}} - n \\
&= U_s + (n-1)K^{1/n} + K\,K^{-n\,-\,1/n} - n \\
&= U_s + (n-1)K^{1/n} + K1 - {}^{n\,-\,1}/n - n \\
&= U_s + (n-1)K^{1/n} + K^{n\,-\,n\,+\,1}/n - n \\
&= U_s + (n-1)K^{1/n} + K^1/n - n \\
&= U_s + nK^{1/n} - K^{1/n} + K^1/n - n \\
&= U_s + n(K^{1/n} - 1)
\end{aligned}
$$

Now, noting that

$$U_s = \frac{C_s}{T_s} = \frac{T_1 - T_s}{T_s} = R_s - 1$$

we have

$$R_s = U_s + 1$$

Thus, K can be rewritten as

$$K = \frac{2}{R_s} = \frac{2}{U_s + 1}$$

and finally

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

Thus, given a set of n periodic tasks and a polling server withg utilization factors U_p and U_s, respectively, the schedulability of the periodic task set is guaranteed under RM if

$$U_p + U_s \leq U_s + n\left(K^{1/n} - 1\right)$$

that is, if

$$U_p \leq n\left[\left(\frac{2}{U_s + 1}\right)^{1/n} - 1\right]$$

Appendix C

U_{lub} for RM + Deferrable Server

To simplify the computation of the bound for n periodic tasks, we first determine the worst-case relations among the tasks, and then we derive the lower bound against the worst-case model.

Consider a set n periodic tasks $(\tau_1, \ldots, \tau_n)$, ordered by increasing periods, and a Deferrable Serve with a higher priority. The worst-case condition for the priodic tasks, is such that $T_1 < T_n < 2T_1$. In the presence of a DS, however, the derivation of the worst-case is more complex and requires the analysis of three different cases. For the sake of clarity, here we analyze one case only, the most general, in which DS may execute three times within the period of the highest-priority periodic task. This happens when DS defers its service at the end of its period and lso executes at the beginning of the next period. In this situation, the full processor utilization is achieved by the following tasks' parameters:

$$\begin{cases} C_s = T_1 - (T_s + C_s) = \dfrac{T_1 - T_s}{2} \\ C_1 = T_2 - T_1 \\ C_2 = T_3 - T_2 \\ \ldots \\ C_{n-1} = T_n - T_{n-1} \\ C_n = T_s - C_s - \displaystyle\sum_{i=1}^{n-1} C_i = \dfrac{3T_s + T_1 - 2T_n}{2} \end{cases}$$

Hence, the resulting utilization is:

$$\begin{aligned} U &= \frac{C_s}{T_s} + \frac{C_1}{T_1} + \cdots + \frac{C_n}{T_n} \\ &= U_s + \frac{T_2 - T_1}{T_1} + \cdots + \frac{T_n - T_{n-1}}{T_{n-1}} + \frac{3T_s + T_1 - 2T_n}{2T_n} \\ &= U_s + \frac{T_2}{T_1} + \cdots + \frac{T_n}{T_{n-1}} + \left(\frac{3T_s}{2T_1} + \frac{1}{2}\right)\frac{T_1}{T_n} - n \end{aligned}$$

defining:

$$\begin{cases} R_s = \dfrac{T_1}{T_s} \\ R_i = \dfrac{T_{i+1}}{T_i} \\ K = \dfrac{1}{2}\left(3\dfrac{T_s}{T_1} + 1\right) \end{cases}$$

and noting that

$$R_1 R_2 \ldots R_{n-1} = \prod_{j=1}^{n-1} R_j = \frac{T_n}{T_1}$$

The utilization factor may be written as

$$U = U_s + \sum_{i=1}^{n-1} R_i + \frac{K}{\prod_{j=1}^{n-1} R_j} - n$$

we minimize U over R_i, $i = 1, \ldots, n-1$. Hence,

$$\begin{aligned}
\frac{\partial U}{\partial R_i} &= \cancelto{0}{\frac{\partial U_s}{\partial R_i}} - \cancelto{0}{\frac{\partial n}{\partial R_i}} + \cancelto{0}{\frac{\partial \sum_{j\neq i}^{n-1} R_j}{\partial R_i}} + \frac{\partial R_i}{\partial R_i} + \frac{\partial K \left(\prod_{j=1}^{n-1} R_j\right)^{-1}}{\partial R_i} \\
&= 1 + K \left(\prod_{j=i}^{n-1} R_j\right)^{-1} \frac{\partial R_i^{-1}}{\partial R_i} \\
&= 1 - K \left(\prod_{j=i}^{n-1} R_j\right)^{-1} R_i^{-2} \\
&= 1 - \frac{K}{R_i^2 \left(\prod_{j\neq i}^{n-1} R_j\right)} \\
&= 1 - \frac{K}{R_i \left(\prod_{j=1}^{n-1} R_j\right)}
\end{aligned}$$

Thus, defining $P = \prod_{j=1}^{n-1} R_j$, U is minimum when:

$$\begin{cases} R_1 P = K \\ R_2 P = K \\ \ldots \\ R_{n-1} P = K \end{cases}$$

that is, when all R_i have the same value:

$$R_1 = R_2 = \cdots = R_{n-1} = K^{1/n}$$

Substituting this value in U we obstain:

$$\begin{aligned}
U_{lub} &= U_s + \sum_{i=1}^{n-1} R_i + \frac{K}{\prod_{j=1}^{n-1} R_j} - n \\
&= U_s + \sum_{i=1}^{n-1} \left(K^{1/n}\right) + \frac{K}{\prod_{j=1}^{n-1} \left(K^{1/n}\right)} - n \\
&= U_s + (n-1)K^{1/n} + \frac{K}{K^{n-1/n}} - n \\
&= U_s + (n-1)K^{1/n} + K\,K^{-n-1/n} - n \\
&= U_s + (n-1)K^{1/n} + K1 - {}^{n-1}/n - n \\
&= U_s + (n-1)K^{1/n} + K{}^{n-n+1}/n - n \\
&= U_s + (n-1)K^{1/n} + K1/n - n \\
&= U_s + nK^{1/n} - K^{1/n} + K1/n - n \\
&= U_s + n(K^{1/n} - 1)
\end{aligned}$$

Now, noting that

$$U_s = \frac{C_s}{T_s} = \frac{T_1 - T_s}{2T_s} = \frac{R_s - 1}{2}$$

we have

$$R_s = 2U_s + 1$$

Thus, K can be rewritten as

$$K = \left(\frac{3}{2R_s} + \frac{1}{2}\right) = \frac{U_s + 2}{2U_s + 1}$$

and finally

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

Thus, given a set of n periodic tasks and a polling server withg utilization factors U_p and U_s, respectively, the schedulability of the periodic task set is guaranteed under RM if

$$U_p + U_s \leq U_s + n\left(K^{1/n} - 1\right)$$

that is, if

$$U_p \leq n\left[\left(\frac{U_s + 2}{2U_s + 1}\right)^{1/n} - 1\right]$$