IN4343 Real-Time Systems

Handling Overload
Dynamic workload

Several applications (e.g., multimedia systems) are characterized by highly variable computational requirements:
Example: phone call

It consists at least of 2 periodic tasks, executed every 20 ms:

- Receives audio signal, decodes it, and transfers packets to the speaker buffer for reproduction.
  \[ C_{i-min} = 1 \text{ ms (silence)} \]
  \[ C_{i-max} = 3 \text{ ms} \]

- Voice sampling, data encoding, speech enhancement, and packet transmission through the modem.
  \[ C_{i-min} = 5 \text{ ms (silence)} \]
  \[ C_{i-max} = 10 \text{ ms} \]

Overall CPU bandwidth: 30-65%
Other causes of overloads

- Optimistic system design (based on average rather than worst-case behavior)
- Malfunctioning of input devices (sensors may send sequence of interrupts in bursts)
- Variations in the environment
- Simultaneous arrivals of events
- Exceptions raised by the kernel
Load definitions

- **Soft aperiodic tasks:** \[ \rho = \lambda \overline{C} \]

- **Hard periodic tasks:** \[ \rho = U = \sum_{i=1}^{n} \frac{C_i}{T_i} \]

- **Generic RT application:**

  if \( g(t_1, t_2) \) is the processor demand in \([t_1, t_2]\), then:

  \[ \rho = \max_{t_1, t_2} \left\{ \frac{g(t_1, t_2)}{t_2 - t_1} \right\} \]
Istantaneous load $\rho(t)$

Does not consider past or future activations, but only current jobs:

\[\tau_1\]
\[\tau_2\]
\[\tau_3\]
stantaneous load $\rho(t)$

Maximum processor demand among those intervals from the current time and the deadlines of all active tasks.

\[
\rho(t) = \max_k \frac{g(t, d_k)}{d_k - t} = \max_k \frac{\sum_{d_i \leq d_k} c_i(t)}{d_k - t}
\]
Example

\[ \tau_1 \]
\[ \tau_2 \]
\[ \tau_3 \]

\[ \rho_1(4) = \frac{2}{4} = 0.5 \]
\[ \rho_2(4) = \frac{5}{6} = 0.83 \]
\[ \rho_3(4) = \frac{7}{9} = 0.78 \]

\[ \rho(4) = 0.83 \]
Examples of load

System designed under worst-case assumptions

System designed under average-case assumptions
Predictability vs. efficiency

Pessimistic assumptions lead to

- high predictability
- low efficiency

Average-case design leads to

- high efficiency
- low predictability
Definitions

**Overrun:** Situation in which a task exceeds its expected utilization.

**Execution Overrun**
The task computation time exceeds its expected value:

![Execution Overrun Diagram]

**Activation Overrun**
The next task activation occurs before its expected time:

![Activation Overrun Diagram]

expected interarrival time
Definitions

**Overload:** Situation in which $\rho(t) > 1$.

**Transient Overload**

$\rho_{\text{max}} > 1$, but $\rho_{\text{avg}} \leq 1$

**Permanent Overload**

$\rho_{\text{avg}} > 1$
Consequences of overruns

deadline misses
(transient overload)

starvation
(permanent overload)

\[ U < 1 \] but sporadic overruns can prevent \( \tau_3 \) to run:
Exercise

• Demonstrate the *domino effect* for EDF
  ➢ give a taskset + overrun(s)

• Demonstrate that RM can result in fewer deadline misses
Providing temporal isolation

Transient overloads can be handled through **Resource Reservation**:

- reserve a fraction of the processor to a set of tasks;
- prevent the task set to use more than the reserved fraction.
Resource Reservation

Resource partition

- $\tau_1$: 20%
- $\tau_2$: 25%
- $\tau_3$: 45%
- $\tau_4$: 10%

Resource enforcement

- A mechanism that prevents a task to consume more than its reserved amount.
- If a task executes more, it is delayed, preserving the resource for the other tasks.

Each task receives a bandwidth $\alpha_i$ and behaves as it were executing alone on a slower processor of speed $\alpha_i$. 
Priorities vs. Reservations

Prioritized Access

τ₁ → P₁
τ₂ → P₂
τ₃ → P₃

Resource Reservation

τ₁ → α₁ = 0.5 → 50%
τ₂ → α₂ = 0.3 → 30%
τ₃ → α₃ = 0.2 → 20%

starvation

τ₁ τ₂ τ₁ τ₂
Benefits of Resource Reservation

1. Resource allocation is easier than priority mapping.

2. It provides temporal isolation: overruns occurring in a reservation do not affect other tasks in the system.
   - Important for modularity and scalability

3. Simpler schedulability analysis:
   - Response times only depends on the application demand and the amount of reserved resource.
Implementing Resource Reservations

Fixed priorities
Dynamic priorities

<table>
<thead>
<tr>
<th>scheduler</th>
<th>server</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM/DM</td>
<td>Sporadic Server</td>
</tr>
<tr>
<td>EDF</td>
<td>CBS</td>
</tr>
</tbody>
</table>
Analysis under Resource Reservations

If a processor is partitioned into $n$ reservations, we must have that:

$$\sum_{i=1}^{n} \alpha_i \leq U_{\text{lub}}^A$$

where $A$ is the adopted scheduling algorithm.
Analysis under Resource Reservations

To describe the time available in a reservation, we need to identify, for any interval \([0, t]\), the minimum time allocated in the worst-case situation.

**Supply function** \(Z(t)\):

*minimum amount of time available in reservation \(R_k\) in every time interval of length \(t\).*
Example: Static time partition

Example of reservation providing 4 units every 10 (bandwidth = 0.4).
Analysis under Resource Reservations

Hence the Processor Demand Criterion can be reformulated as follows:

\[ \forall t > 0, \quad dbf(t) \leq Z(t) \]
Analysis under Resource Reservations

A simpler, but only sufficient test, can be derived by replacing the supply function $Z(t)$ with a lower bound, called supply bound function $sbf(t)$:

$$\forall t > 0, \quad dbf(t) \leq sbf(t)$$
Supply bound function

A supply bound function has the following form:

\[ sbf(t) = \max\{0, \alpha(t - \Delta)\} \]

\[ \alpha = \text{bandwidth} \]
\[ \Delta = \text{service delay} \]
Supply bound function

For a given supply function $Z(t)$, the bandwidth $\alpha$ and the delay $\Delta$ can be formally defined as follows:

\[
\alpha = \lim_{t \to \infty} \frac{Z(t)}{t}
\]

\[
\Delta = \sup_{t \geq 0} \left\{ t - \frac{Z(t)}{\alpha} \right\}
\]
Example: Periodic Server

For a periodic server with budget $Q_s$ and period $P_s$ running at the highest priority, we have:

$$\alpha = \frac{Q_s}{P_s}$$

$$\Delta = P_s - Q_s$$
Example: Periodic Server

For a periodic server with budget $Q_s$ and period $P_s$, running at unknown priority, we have:

\[ \alpha = \frac{Q_s}{P_s} \]

\[ \Delta = 2(P_s - Q_s) \]
Handling permanent overload

Permanent overload conditions can be handled using two different approaches:

- **Value-based scheduling**
  - least importance tasks are rejected
  - important tasks receive full service

- **Performance degradation**
  - All tasks are executed
  - but with reduced requirements
Value-based scheduling

- If $\rho > 1$, no all tasks can finish within their deadline.

- To avoid domino effects, the load is reduced by rejecting the least important tasks.

- To do that, the system must be able to handle tasks with both timing constraints and importance values.
Deadline and Value

- Under RM and EDF, the value of a task is implicitly encoded in its period or deadline.

- However, in a chemical plant controller, a task reading the steam temperature every 10 seconds is more important than a task which updates the clock icon every second.
How to assign values

A task $\tau_i$ can be assigned a value $v_i$ according to different criteria. Those most common are:

| $v_i = V_i$ | arbitrary constant |
| $v_i = C_i$ | computation time |
| $v_i = V_i/C_i$ | value density |
Value as a function of time

In a real-time system, the value of a task depends on its completion time and criticality:

- **Hard**
  - Value function $v_i (f_i)$
  - Completion time $d_i$
  - Due日期 $r_i$

- **Soft**
  - Value function $v_i (f_i)$
  - Completion time $d_i$
  - Due date $r_i$

- **Firm**
  - Value function $v_i (f_i)$
  - Completion time $d_i$
  - Due date $r_i$

- **Target Sensitive**
  - Value function $v_i (f_i)$
  - Completion time $d_i$
  - Due date $r_i$
Performance evaluation

- The performance of a scheduling algorithm A on a task set T can be evaluated through its **Cumulative Value**:

\[
\Gamma_A(T) = \sum_{i=1}^{n} v_i(f_i)
\]

- Note that, under overload conditions:

\[
\Gamma_A(T) < \Gamma_{\text{max}}(T) = \sum_{i=1}^{n} V_i
\]
Optimality under overloads

If

\[ \Gamma^*(T) = \max_A \Gamma_A(T) \]

the performance of an algorithm can be evaluated with respect to \( \Gamma^* \).

In overload conditions, there are no optimal on-line algorithms able to guarantee a cumulative value equal to \( \Gamma^* \).
Proof \hspace{0.5cm} (assume: \( V_i = C_i \))

To maximize \( \Gamma_A \) we should know the future.

If at time \( t = 0 \), \( r_3 \) is not known, we cannot select the task that maximizes the cumulative value.
Competitive Factor

- Let $\Gamma^*$ be the maximum cumulative value achievable by an optimal clairvoyant algorithm.

- An algorithm $A$ has a competitive factor $\varphi_A$, if it is guaranteed that, for any task set, it achieves:

$$\Gamma_A \geq \varphi_A \Gamma^*$$

- Hence, $\varphi_A \in [0,1]$ and can be computed as:

$$\varphi_A = \min_T \frac{\Gamma_A(T)}{\Gamma^*(T)}$$
Competitive factor of EDF

- It is easy to show that $\varphi_{\text{EDF}} = 0$:

\[ \tau_1 \quad \tau_2 \]

\[ V_1 = K \]

\[ V_2 = \varepsilon K \]

In such a situation, $\Gamma_{\text{EDF}} = V_2$ and $\Gamma^* = V_1$, hence $\Gamma_{\text{EDF}} / \Gamma^* = V_2 / V_1 \to 0$ for $V_1 \gg V_2$
A theoretical upper bound

[Baruah et al., 91]

If $\rho \geq 2$ and task value is proportional to computation time, then no on-line algorithm can have a competitive factor greater than 0.25.

That is: $\max_A \varphi_A \leq 0.25$
Proof by adversary argument

A scheduling game

- In order to win, the adversary generates tasks as a function of the player's choices.
- At the end, each one shows the cumulative value.
Exercise

4 min

• Play the adversary ...
  and beat EDF (or any other policy)

• Hints:
  ➢ load ($\rho$) $\geq 2$ — allow for (bad) choices
  ➢ avoid slack — limit options / predictable “moves”
  ➢ vary task sizes — maximize gain / loss
Task generation strategy

- The adversary generates 2 types of task:
  - **Major tasks:** $C_i = D_i$, $r_{i+1} = d_i - \varepsilon$
  - **Associated tasks:** $C_i = \varepsilon$, $r_{i+1} = d_i$

(note: $p = 2$)
Task generation strategy

- If the player decides to abort a major task in favor of an associated task, the adversary interrupts the sequence of associated tasks.
Task generation strategy

- If the player decides to complete $\tau_i$, the game terminates with the generation of $\tau_{i+1}$.

\[
\begin{align*}
\Gamma_{on} &= C_i \\
\Gamma^{*} &= \sum_{j=0}^{i+1} C_j
\end{align*}
\]
Task generation strategy

- Since the overload must have a finite duration, the game terminates with $\tau_m$ (with $m$ finite).

NOTE: The player can complete at most one task:

- if it schedules an associated task, it gets $\Gamma_{on} = \varepsilon$
- if it schedules a major task, it gets $\Gamma_{on} = C_i$

Viceversa, the adversary can accumulate:

$$\Gamma^* = \sum_{j=0}^{i+1} C_j$$

- either executing all associated tasks
- or alternating major and associated tasks.
Critical sequence

- Let $\tau_0, \tau_1, \tau_2, \ldots, \tau_i, \tau_{i+1}, \ldots, \tau_m$ be the longest sequence of major tasks generated by the adversary.

- If the player schedules $\tau_i$, it gets:

$$
\Gamma_{on} = C_i \quad \& \quad \Gamma^* = \sum_{j=0}^{i+1} C_j \quad \Rightarrow \quad \phi_{on}(i) = \frac{C_i}{\sum_{j=0}^{i+1} C_j}
$$

- To create problems to the player the next task must be generated so that: $\phi_{on}(i+1) \leq \phi_{on}(i)$;

- On the other hand, to convince the player to continue, it must be: $\phi_{on}(i+1) \geq \phi_{on}(i)$;
Critical sequence

- Hence, the worst possible sequence for the player is such that: \( \varphi_{\text{on}}(i+1) = \varphi_{\text{on}}(i) \).

- Be \( 1/k \) such a value. Then, if \( C_0 = 1 \), the worst-case sequence must be such that:

\[
\frac{C_i}{\sum_{j=0}^{i+1} C_j} = \frac{1}{k}
\]

That is:

\[
\begin{cases}
  C_0 = 1 \\
  C_{i+1} = kC_i - \sum_{j=0}^{i} C_j
\end{cases}
\]
• If the player decides to schedule $\tau_m$ we have:

$$\Gamma_{on} = C_m \quad \& \quad \Gamma^* = \sum_{j=0}^{m} C_j \quad \Rightarrow \quad \varphi_{on}(m) = \frac{C_m}{\sum_{j=0}^{m} C_j}$$

• Thus, if there exists an $m$ such that

$$\varphi_{on}(m) \leq \frac{1}{k}$$

we can claim that

$$\varphi_{on}(i) \leq \frac{1}{k} \quad \forall i$$

which means that the competitive factor of a on-line algorithm cannot be greater than $1/k$. 
• Note that $\varphi_{on}(m)$ is equal to:

$$\frac{C_m}{\sum_{j=0}^{m} C_j} = \frac{C_m}{\sum_{j=0}^{m-1} C_j + C_m} = \frac{C_m}{\sum_{j=0}^{m-1} C_j + (kC_{m-1} - \sum_{j=0}^{m-1} C_j)} = \frac{C_m}{kC_{m-1}}$$

• Hence, condition $\varphi_{on}(m) \leq 1/k$ is equivalent to:

$$\frac{C_m}{kC_{m-1}} \leq \frac{1}{k} \quad \Rightarrow \quad C_m \leq C_{m-1}$$
It is possible to prove that the sequence:

\[
\begin{align*}
C_0 &= 1 \\
C_{i+1} &= kC_i - \sum_{j=0}^{i} C_j
\end{align*}
\]

diverges for \(k \geq 4\), that is \(\nexists m\) such that \(C_m \leq C_{m-1}\)

Instead, for \(k < 4\), \(\exists m\) such that \(C_m \leq C_{m-1}\)

hence, the player can never get \(\varphi_{on} > 1/4\).
A hint at the proof

Check the book for details

\[
\begin{align*}
C_0 &= 1 \\
C_{i+1} &= kC_i - \sum_{j=0}^{i} C_j \\
C_0 &= 1 \\
C_1 &= k - 1 \\
C_{i+2} &= k(C_{i+1} - C_i)
\end{align*}
\]

\[x^2 - kx + k = 0\]

\[C_i = d_1(x_1)^i + d_2(x_2)^i\]

\[x = \frac{k \pm \sqrt{k^2 - 4k}}{2}\]
In general, the upper bound of the competitive factor is a function of the load and varies as follows:

\[ \phi_{on} = \sqrt{4/27} \approx 0.385 \]
Value-based scheduling

Is it all that bad?

Competitive factor

• EDF = 0.0
• $0.25 \leq \text{online} \leq 0.385$

But bounds are for unrealistic conditions

• zero laxity
• permanent overload
• hugely different task execution times

In practice RM/EDF (with modifications) do fine
What this lecture is about

What can we do when the system is permanently overloaded?

We will study six methods
Existing techniques

Value-based scheduling
- Best-Effort Scheduling
- Admission Control
- Robust Scheduling

Performance degradation
- Job Skipping
- Reduce computation
- Period Adaptation

be nice … and accept all
be picky … and drop some
be sneaky … and cut resources
Best-effort scheduling

- Tasks are always accepted in the system.
- Performance is controlled through a suitable (value-based) priority assignment.
- **Problem:** domino effect.
Every task is subject to an acceptance test which keeps the load \( \leq 1 \). Check schedulability too.

It prevents domino effects, but does not take values into account.

Low efficiency due to the worst-case guarantee (tasks may be unnecessarily rejected).
Best Effort Scheduling

... does not control load, but cares about value

Admission Control

... does not care about value, but controls load

We need something that considers both, value and control
Robust scheduling

- Task scheduling and task rejection are controlled by two separate policies.
- Tasks are scheduled by deadline, rejected by value.
- In case of early completions, rejected tasks can be recovered by a reclaiming mechanism.
What method should we use for scheduling?

we ❤️ EDF

Robust EDF

- Scheduling Policy $\Rightarrow$ EDF

- Rejection policy
  when an overload is detected, reject the least value task which can bring the load below 1.

- Recovery policy
  - keep rejected tasks by decreasing values;
  - when there is enough spare time, re-accept the highest value task which is still feasible.
Example: task rejection

What task should we reject?

Reject the least value task that can bring the load below 1
Example: task rejection

At time $t = 4 \implies \tau_3$ rejected
Example: task recovery

\begin{align*}
\tau_1 & \quad 7 \\
\tau_2 & \quad 10 \\
\tau_3 & \quad 5 \\
\tau_4 & \quad 2 \\
\tau_5 & \quad 3 \\
\end{align*}

at time $t = 8 \quad \Rightarrow \quad \tau_3$ can be recovered
Existing techniques

Value-based scheduling
- Best-Effort Scheduling
- Admission Control
- Robust Scheduling

Performance degradation
- Job Skipping
- Reduce computation
- Period Adaptation
Performance Degradation

The load can be decreased not only by rejecting tasks, but also by reducing their performance requirements. This can be done by:

1) Job Skipping

2) Reduce Computation

3) Period Adaptation

\[ U_i = \frac{C_i}{T_i} \]

\[ \text{reduce } C_i \]

\[ \text{increase } T_i \]
Job skipping

Periodic load can also be reduced by skipping some jobs, once in a while.

Many systems tolerate skips, if they do not occur too often:

- multimedia systems (video reproduction)
- inertial systems (robots)
- monitoring systems (sporadic data loss)
Example

The system is overloaded, but tasks can be schedulable if $\tau_1$ skips one instance every 3:

$$U_p = \frac{1}{2} + \frac{4}{6} = 1.17 > 1$$
FIRM task model

- Every job can either be executed within its deadline, or completely rejected (skipped).

- A percentage of task instances must be guaranteed off line to finish in time.

- Each task $\tau_i$ is described by $(C_i, T_i, D_i, S_i)$:
  
  $S_i$ is the minimum distance between two consecutive skips.
• Every instance can be red or blue:
  – red instances must finish within their deadline
  – blue instances can be aborted

• If a blue instance is aborted, the next $S_{i-1}$ instances must be red.

• If a blue instance is completed within its deadline, the next instance is still blue.
Example

\[ C_i = 1 \quad T_i = 2 \quad D_i = 2 \quad S_i = 3 \]
Deeply-red systems

Use of skips is NP-hard and not optimal when working under RM and EDF, except under deeply-red conditions.

Conditions for an optimal deeply-red system

1) all tasks are synchronously activated
2) first $S_i-1$ instances of every task are red
Schedulability Analysis

The old good processor demand approach, but with a twist

\[ g_i(0, L) = \left( \left\lfloor \frac{L}{T_i} \right\rfloor - \left\lfloor \frac{L}{T_iS_i} \right\rfloor \right)C_i \]

**Sufficient condition**

A set of periodic tasks is schedulable by EDF if

\[ \forall L \geq 0 \quad \sum_{i=1}^{n} \left( \left\lfloor \frac{L}{T_i} \right\rfloor - \left\lfloor \frac{L}{T_iS_i} \right\rfloor \right)C_i \leq L \]

Question: how long should \( L \) be? (no answer in the book)
A necessary condition

Theorem: A set of firm periodic tasks is not schedulable if

\[ \sum_{i=1}^{n} \frac{C_i (S_i - 1)}{T_i S_i} > 1 \]

NOTE: the sum represents the utilization of the computation that must take place.
Existing techniques

Value-based scheduling
- Best-Effort Scheduling
- Admission Control
- Robust Scheduling

Performance degradation
- Job Skipping
- Reduce computation
- Period Adaptation

$$U_i = \frac{C_i}{T_i}$$
Can we really start cutting the Ci’s just like that?

Yes we can, Ci’s are not black and white.

There are shades of gray
Reducing precision

In many applications, computation can be performed at different level of precision: the higher the precision, the longer the computation. Examples are:

- binary search algorithms
- image processing and computer graphics
- neural learning
Sensitivity Analysis

Example:
\[
\begin{align*}
D_1 &= 5, \quad T_1 = 10 \\
D_2 &= 8, \quad T_2 = 10
\end{align*}
\]

Feasibility region:
\[
\begin{align*}
C_1 + C_2 &\leq D_2 \\
C_1 &\leq D_1
\end{align*}
\]

Direction: \( \Delta C_1 = \Delta C_2 \)

New values:
\[
\begin{align*}
C_1 &= 3 \\
C_2 &= 5
\end{align*}
\]
Sensitivity Analysis

1. Identify the schedulability region of the application in the space of computation times (C-space).

2. Identify the application in the C-space;

3. Define a direction for decreasing requirements;

4. Compute the new computations values;

5. Reduce the code duration accordingly.
Reducing computation times is fine

but

we need a more structured manner to reduce them
Imprecise computation

In this model, each task $\tau_i (C_i, D_i, w_i)$ is divided in two portions:

- a **mandatory** part: $\tau^m_i (M_i, D_i)$
- an **optional** part: $\tau^o_i (O_i, D_i)$

$w_i$ is an importance weight
Imprecise computation

In this model, a schedule is said to be:

- **feasible**, if all mandatory parts complete in $D_i$
- **precise**, if also the optional parts are completed.

**error:** $\varepsilon_i = O_i - \sigma_i$

**average error:** $\varepsilon_a = \sum_{i=1}^{n} w_i \varepsilon_i$

**GOAL:** minimize the average error
Existing techniques

Value-based scheduling

- Best-Effort Scheduling
- Admission Control
- Robust Scheduling

Performance degradation

- Job Skipping
- Reduce computation
- Period Adaptation

\[ U_i = \frac{C_i}{T_i} \]

increase \( T_i \)
Relaxing timing constraints

- The idea is to reduce the load by increasing deadlines and/or periods.
- Each task must specify a range of values in which its period must be included.
- Periods are increased during overloads, and reduced when the overload is over.

Many control applications require tasks running at variable rates, to cope with changing conditions.
Examples: altimeter reading

- The smaller the altitude, the higher the acquisition rate:
Obstacle avoidance

- The closer the obstacle, the higher the acquisition rate:
**Example**

<table>
<thead>
<tr>
<th>task</th>
<th>$C_i$</th>
<th>$T_{i0}$</th>
<th>$T_{min}$</th>
<th>$T_{max}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau_1$</td>
<td>10</td>
<td>20</td>
<td>20</td>
<td>25</td>
</tr>
<tr>
<td>$\tau_2$</td>
<td>10</td>
<td>40</td>
<td>40</td>
<td>50</td>
</tr>
<tr>
<td>$\tau_3$</td>
<td>15</td>
<td>70</td>
<td>35</td>
<td>80</td>
</tr>
</tbody>
</table>

$$U_p = \frac{10}{20} + \frac{10}{40} + \frac{15}{70} = 0.96$$
Load adaptation

If a new task $\tau_4$ arrives with: $C_4 = 5$, $T_4 = 30$ the system is not schedulable any more:

$$U_p = \frac{10}{20} \frac{10}{40} \frac{15}{70} \frac{5}{30} = 1.13$$

However, there exists a feasible schedule within the specified ranges:

$$U_p = \frac{10}{23} \frac{10}{50} \frac{15}{80} \frac{5}{30} = 0.99$$
There are an infinite number of Ti’s combinations

we need a structured method to achieve an efficient and fair reduction among all tasks
Elastic task model

- Tasks’ utilizations are treated as elastic springs and can be changed by period variations.

- The resistance of a task to a period variation is controlled by an elastic coefficient $E_i$:

  $\Rightarrow$ the greater $E_i$, the greater the elasticity
Elastic task model

- A periodic task $\tau_i$ is characterized by:
  \[ (C_i, T_{i0}, T_{i-min}, T_{i-max}, E_i) \]
- The actual period $T_i \in [T_{i-min}, T_{i-max}]$
Special cases

- A task with $T_{\text{min}} = T_{\text{max}}$ is equivalent to a hard task.

- A task with $E_i = 0$ can intentionally change its period but does not allow the system to do that.
Compression algorithm

During overloads, utilizations must be compressed to bring the load below one.
The linear spring analogy

\[ F = k_1(x_{1o} - x_1) \]
\[ F = k_2(x_{2o} - x_2) \]
\[ F = k_3(x_{3o} - x_3) \]

\[ x_1 + x_2 + x_3 = L_d \]
\[ x_{1o} + x_{2o} + x_{3o} = L_0 \]
Solution without constraints
(min $x_i$ can be zero)

Summing the equations, we have:

$$F\left(\frac{1}{k_1} + \frac{1}{k_2} + \frac{1}{k_3}\right) = (x_{1o} + x_{2o} + x_{3o}) - (x_1 + x_2 + x_3)$$

$$= (L_0 - L_d)$$

That is:

$$F = \frac{(L_0 - L_d)}{\frac{1}{k_1} + \frac{1}{k_2} + \frac{1}{k_3}}$$
Solution without constraints

Substituting $F$ in the equations, we have:

$$F = k_1 (x_{1o} - x_1) = \frac{(L_0 - L_d)}{\frac{1}{k_1} + \frac{1}{k_2} + \frac{1}{k_3}}$$

That is:

$$x_1 = x_{1o} - (L_0 - L_d) \frac{1}{\frac{1}{k_1} + \frac{1}{k_2} + \frac{1}{k_3}}$$

$E_i = \frac{1}{k_i}$

$E_s = \sum_{i=1}^{n} E_i$
Period computation

\[ U_i = U_{io} - (U_0 - U_d) \frac{E_i}{E_s} \]

And then:

\[ T_i = \frac{C_i}{U_i} \]
Exercise (5 min)

Use the Elastic Period Method to derive the periods that guarantee the schedulability of these tasks under Rate Monotonic scheduling (LL bound)

<table>
<thead>
<tr>
<th></th>
<th>$C_i$</th>
<th>$T_{i \text{min}}$</th>
<th>$T_{i \text{max}}$</th>
<th>$E_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>5</td>
<td>8</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>t2</td>
<td>6</td>
<td>10</td>
<td>25</td>
<td>3</td>
</tr>
</tbody>
</table>
Solution with constraints

\[(\min x_i \text{ greater than zero})\]

Iterative solution:
Period computation

The set $\Gamma$ can be divided into two subsets:

- a set $\Gamma_f$ of fixed springs having minimum length;
- a set $\Gamma_v$ of variable springs that can still be compressed.

\[
\forall \tau_i \in \Gamma_v \quad U_i = U_{io} - (U_{v0} - U_d + U_f) \frac{E_i}{E_v}
\]

\[
U_{v0} = \sum_{\tau_i \in \Gamma_v} U_{io} \quad U_f = \sum_{\tau_i \in \Gamma_f} U^\text{min}_i \quad E_v = \sum_{\tau_i \in \Gamma_v} E_i
\]

If for some task $U_i < U^\text{min}_i$, then set $U_i = U^\text{min}_i$, update $\Gamma_v$ and $\Gamma_f$ and repeat the process.
Experimental results

Overload handling due to a new task arrival:

\[ \tau_4 \text{ arrives at time } t^* \]
Experimental results

Overload handling due to an increased rate:

\( \tau_1 \) increases its rate at \( t^* \)
Existing techniques

Value-based scheduling

- Best-Effort Scheduling
- Admission Control
- Robust Scheduling

Performance degradation

- Job Skipping
- Reduce computation
- Period Adaptation