Load Control Methods

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

Instantaneous load ρ(t)

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

$$\rho(t) = \max_k \frac{g(t,d_k)}{d_k - t} = \max_k \frac{\sum_{d_i \le d_k} c_i(t)}{d_k - t}$$

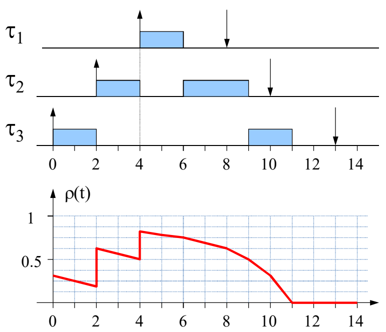
Instantaneous load ρ(t)

We can have at most one deadline for each active task τ_k, hence:

$$\rho_k(t) = \frac{\sum_{d_i \le d_k} c_i(t)}{d_k - t}$$

$$\rho(t) = \max_k \rho_k(t)$$

Example



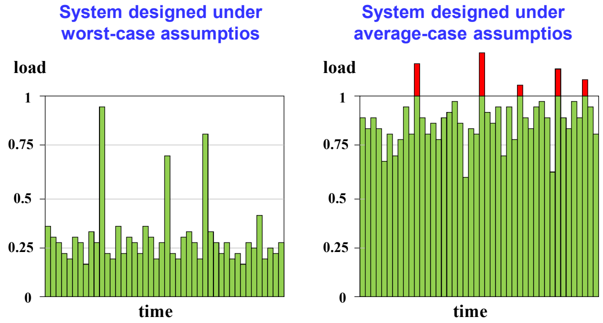
$\rho_1(4) = 2/4 = 0.5$

$\rho_2(4) = 5/6 = 0.83$

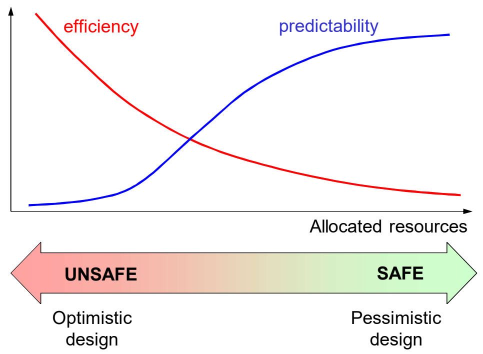
$\rho_3(4) = 7/9 = 0.78$

$\boxed{\rho(4) = 0.83}$

Load and design assumptions



Predictability vs. efficiency



A matter of cost

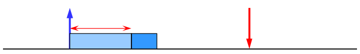
- High predictability and low efficiency means wasting resources $\Rightarrow$ **high cost**
  - it can be justified only for very critical systems
- High efficiency in resource usage requires a the system to:
  - handle and tolerate overloads
  - adapt for graceful degradation
  - plan for exception handling mechanisms

Definitions

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

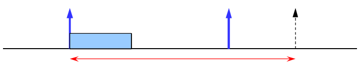
Execution Overrun

The task computation time exceeds its expected value:



Activation Overrun

The next task activation occurs before its expected time:



Definitions

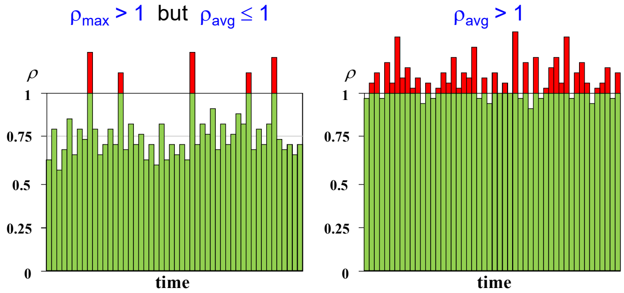
Overload: Situation in which $\rho(t) > 1$.

Transient Overload

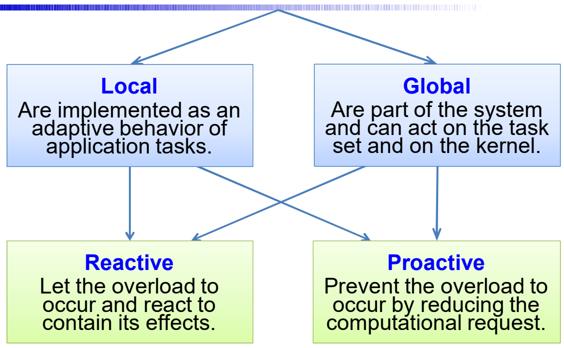
$\rho_{max} > 1$ but $\rho_{avg} \leq 1$

Permanent Overload

$\rho_{avg} > 1$

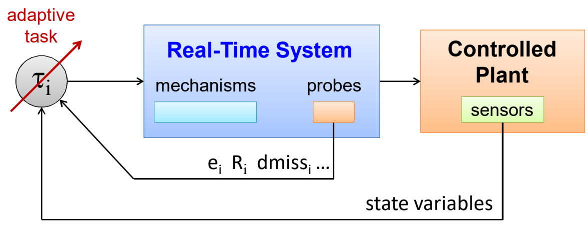


Overload handling methods



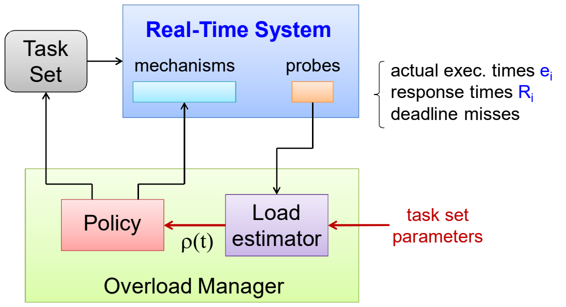
Local Policies

They are implemented inside specific application tasks as an adaptive behavior to reduce their computational demand:



Global Policies

They are implemented as a component (Overload Manager) that can operate both on the kernel and on the entire task set:



Reactive methods

They let the overload to occur, detect it and react with a proper action aimed at containing its effects.

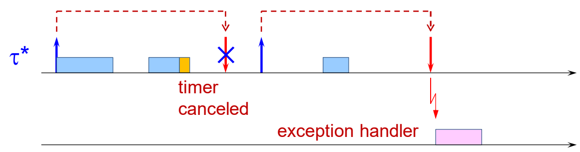
They must handle three phases:

1. Overload detection: this is done by proper kernel probes and timers for catching deadline misses, execution overruns, etc.
2. Exception notification: this is done by generating an interrupt for the kernel and a message for the user.
3. Exception handling: depending on the criticality of the exception, possible actions are:
   - system reset
   - abortion of the running task
   - rejection of least important tasks
   - performance degradation
   - no action: just notification

Example of reactive approach

If τ^* is a critical task that has to finish by a deadline d, a timer can be set at its activation to interrupt after d.

If the task finishes before d, the timer is canceled, otherwise an exception is raised:



In the worst case, the exception handler can reset the system.

Proactive methods

They prevent the overload to occur by proper admission tests and by reducing the computational demand of the application.

The computational demand can be reduced by:

- rejecting tasks
- reducing computation
- reducing priority (under fixed priority systems)
- postponing the deadline (under deadline-based systems)
- skipping specific jobs
- reducing the activation rate of periodic tasks

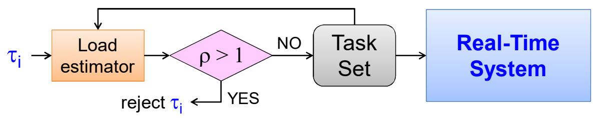
Existing proactive approaches

Depending on the type of performed action, the following proactive approaches can be distinguished:

- **Admission control methods**
  The load is reduced by rejecting one or more tasks.
- **Performance degradation methods**
  The load is reduced by degrading the system performance acting on the task set parameters (computation times, periods, or skipping specific jobs).

Simple admission control

The load is estimated at every task activation based on worst-case task set parameters: if $\rho \leq 1$, the task is accepted, otherwise it is rejected:

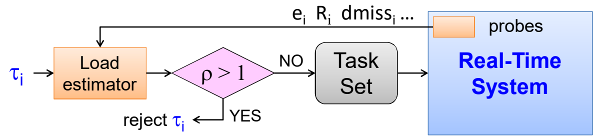


Pros: Simplicity (like the telephone system).
Accepted tasks are guaranteed to receive full service.

Cons: Load estimation is pessimistic (using WCETs and MITs) so tasks could be unnecessarily rejected.
Task importance is not taken into account.

Admission by feedback

The load is estimated at every task activation based on actual execution behavior detected by kernel probes [Stankovic, '99]:



Pros: It increases system efficiency, accepting more tasks.

Cons: Tasks can experience deadline misses (not good for safety-critical systems).
Task importance is not taken into account.

Performance degradation

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

- Degrading functionality (reducing task code)
- Skipping specific jobs
- Increasing periods

Functional degradation

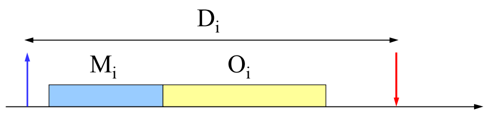
In many applications, computation can be performed at different level of precision: the higher the precision, the longer the computation. Examples are:

- Binary search algorithms
- Image processing and computer graphics
- Neural learning
- Any time control

Imprecise computation

In this model, each task τ_i (C_i, D_i, w_i) is divided in two parts:

- a **mandatory** part: τ^m_i (M_i, D_i)
- an **optional** part: τ^o_i (O_i, D_i)
- $C_i = M_i + O_i$
- w_i = importance weight

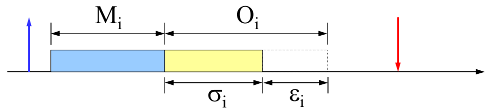


Imprecise computation

In this model, a schedule is said to be:

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

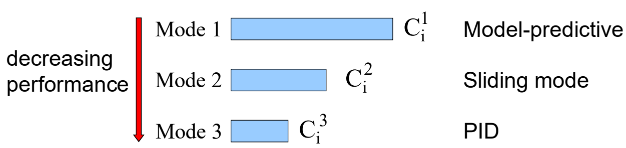
error: $\varepsilon_i = O_i - \sigma_i$ **average error:** $\varepsilon_a = \sum_{i=1}^{n} w_i \varepsilon_i$



GOAL: minimize the average error

Multiple versions

If a task does not comply with the imprecise computational model, another option is to implement a function in multiple versions (operational modes):



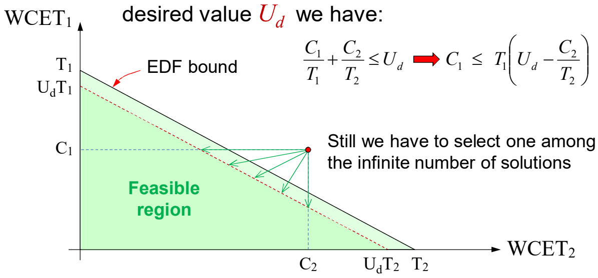
Sensitivity Analysis

Given an unschedulable task set, the problem is:

Which C_i's should be changed and how much?

If we require the total utilization to be equal to a desired value U_d we have:

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} \le U_d \;\Rightarrow\; C_1 \le T_1\left(U_d - \frac{C_2}{T_2}\right)$$



Still we have to select one among the infinite number of solutions

Functional degradation

This method can be implemented both:

- globally: if the mode is selected by the overload manager
- locally: if the mode is selected by the task itself

Example of **local overload management**

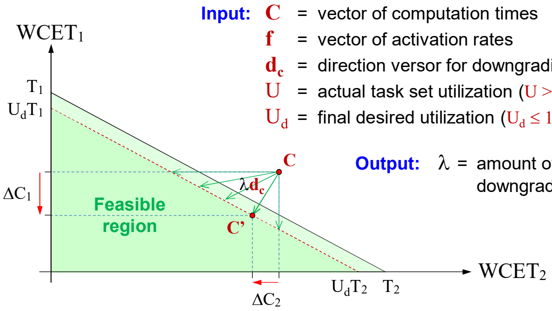
```
while (1) {          // periodic loop
      i = 0;         // modes m ∈ [1,M]
      do {           // select the best feasible mode
           i = i + 1;
           rho = estimate_load(mode[i]);
      } while ((rho > 1) && (i < M));

      if (rho > 1) exception(UNFEASIBLE);

      execute(mode[i]);
      wait_for_next_period();
}
```

Global methods

Global methods can find the optimal solution taking task constraints into account:

Input:
- $\mathbf{C}$ = vector of computation times
- $\mathbf{f}$ = vector of activation rates
- $\mathbf{d}_c$ = direction versor for downgrading $\mathbf{C}$
- U = actual task set utilization ($U > 1$)
- U_d = final desired utilization ($U_d \le 1$)

Output: λ = amount of downgrading

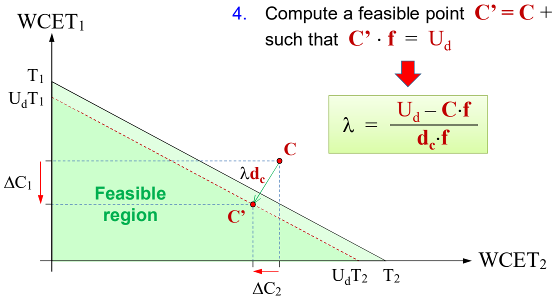


Global methods

General approach:

1. Given computation times $\mathbf{C}$ and rates $\mathbf{f}$
2. Select a direction $\mathbf{d}_c$ for downgrading $\mathbf{C}$
3. Set a desired utilization U_d
4. Compute a feasible point $\mathbf{C'} = \mathbf{C} + \lambda \mathbf{d}_c$ such that $\mathbf{C'} \cdot \mathbf{f} = U_d$

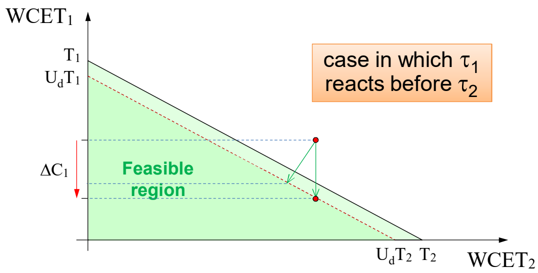
$$\lambda = \frac{U_d - \mathbf{C} \cdot \mathbf{f}}{\mathbf{d}_c \cdot \mathbf{f}}$$



Local methods

Local methods cannot find the optimal solution and can downgrade more than needed.

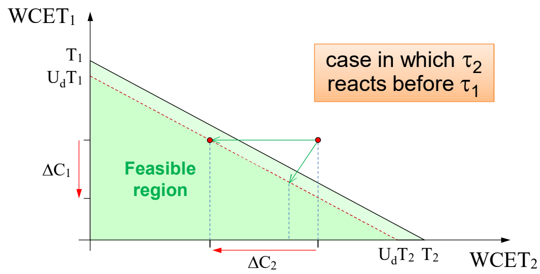
case in which τ_1 reacts before τ_2



Local methods

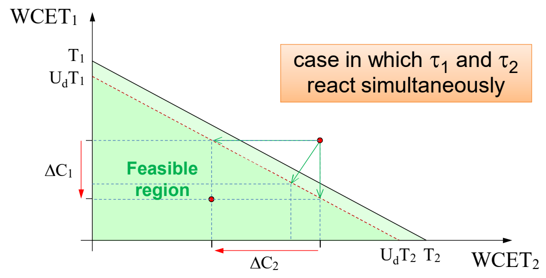
Local methods cannot find the optimal solution and can downgrade more than needed.

case in which τ_2 reacts before τ_1



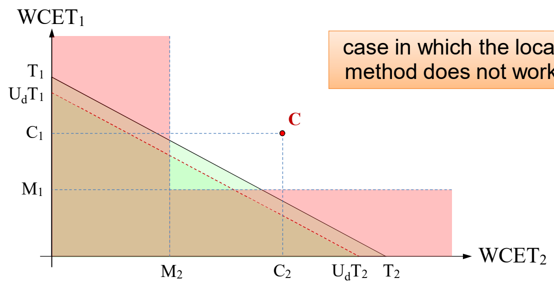
Local methods

Local methods cannot find the optimal solution and can downgrade more than needed.



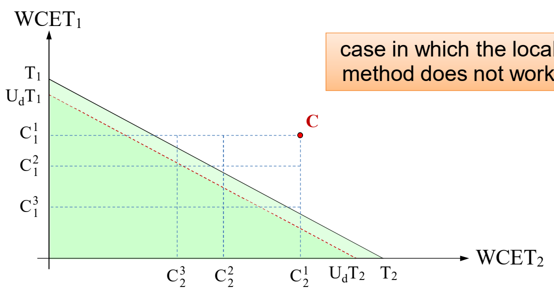
Imprecise computation

The variability range is continuous but limited by the mandatory parts.



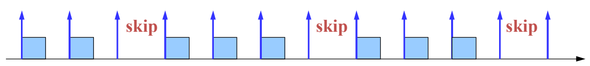
Multiple versions

The variability range is larger, but discrete.



Job skipping

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



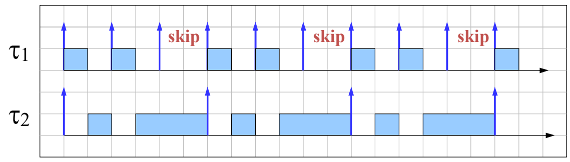
Many systems tolerate skips, if they do not occur too often:

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

Example

The system is overloaded, but tasks can be schedulable if τ_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 τ_i is described by (C_i, T_i, D_i, S_i):

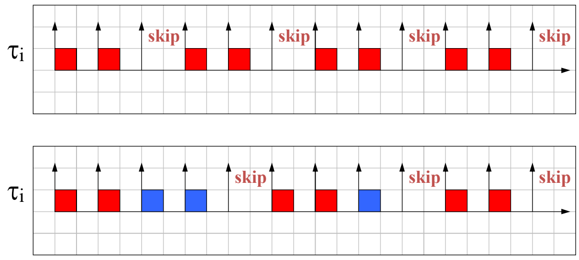
  S_i is the minimum number of jobs that must be executed between two consecutive skips.

FIRM task model

- 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**.
- The first $S_i - 1$ instances of every task must be **red**.

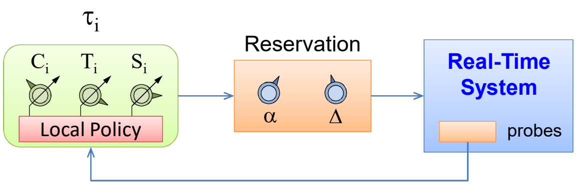
Example

$C_i = 1 \quad T_i = 2 \quad D_i = 2 \quad S_i = 3$



Local adaptation

A local adaptation approach is also possible for a task to comply with the assigned reservation:



Equivalent utilization factor

$$U_p^* = \max_{L \geq 0} \left\{ \frac{\sum_{i=1}^{n} g_i(0,L)}{L} \right\}$$

$$g_i(0,L) = \left(\left\lfloor \frac{L}{T_i} \right\rfloor - \left\lfloor \frac{L}{T_i S_i} \right\rfloor \right) C_i$$

Schedulability Analysis

A sufficient condition

Theorem: *A set of firm periodic tasks is schedulable by EDF if*

$$U_p^* \leq 1$$

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.

Relaxing timing constraints

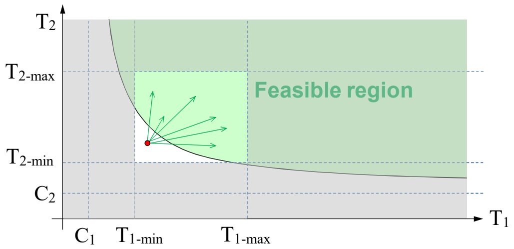
- The idea is to reduce the load by increasing task periods.
- Each task must specify a period range $[T_{min}, T_{max}]$ compatible with its function.
- Periods are increased during overloads, and reduced when the overload is over.

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

Lot of feasible solutions

In general, there can be a lot of feasible solutions with periods inside the specified range, the problems is:

How do we select a solution among all feasible ones?

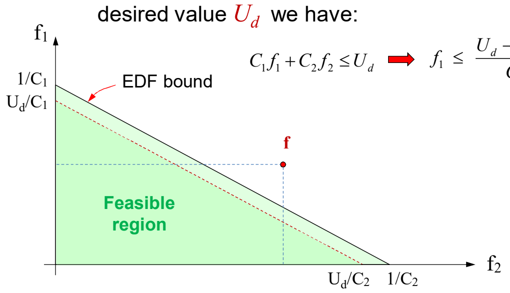


Sensitivity Analysis

We can follow the same approach used for reducing computation times, in the rate-space:

If we require the total utilization to be equal to a desired value U_d we have:

$$C_1 f_1 + C_2 f_2 \le U_d \Rightarrow f_1 \le \frac{U_d - C_2 f_2}{C_1}$$



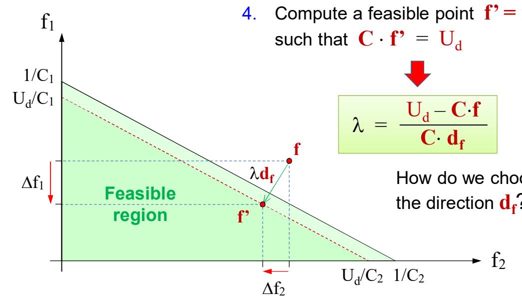
Sensitivity Analysis

General approach:

1. Given computation times $\mathbf{C}$ and rates $\mathbf{f}$
2. Select a direction $\mathbf{d_f}$ for downgrading $\mathbf{f}$
3. Set a desired utilization U_d
4. Compute a feasible point $\mathbf{f'} = \mathbf{f} + \lambda \mathbf{d_f}$ such that $\mathbf{C} \cdot \mathbf{f'} = U_d$

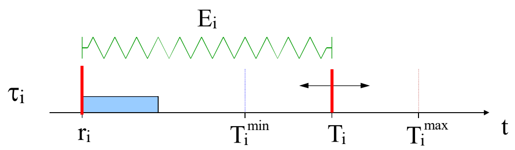
$$\lambda = \frac{U_d - \mathbf{C} \cdot \mathbf{f}}{\mathbf{C} \cdot \mathbf{d_f}}$$

How do we choose the direction $\mathbf{d_f}$?



Elastic task model

- Tasks' utilizations are treated as elastic springs and can be changed by period variations.
- The flexibility of a task to a period variation is controlled by an **elastic coefficient E_i** (the higher E_i the greater the elasticity).
- A periodic task τ_i is characterized by: $(C_i, T_i^{min}, T_i^{max}, E_i)$



Special cases

- A task with $T^{min} = T^{max}$ 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.

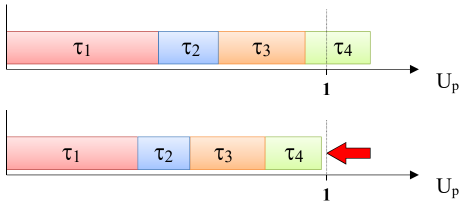
Definitions

$$U_i^{max} = \frac{C_i}{T_i^{min}} \qquad U^{max} = \sum_{i=1}^{n} U_i^{max}$$

$$U_i^{min} = \frac{C_i}{T_i^{max}} \qquad U^{min} = \sum_{i=1}^{n} U_i^{min}$$

Compression algorithm

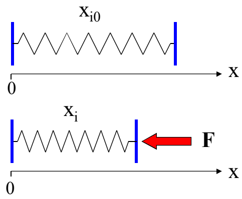
During overloads, utilizations must be compressed to bring the load below one.



The spring analogy

An elastic task can be compared with a linear spring:

spring length ⇔ task utilization



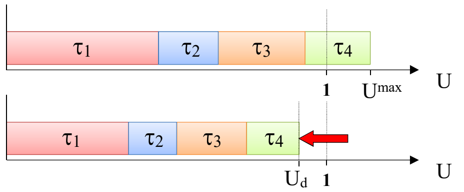
$$\begin{cases} x_{i0} \longrightarrow U_i^{max} = \dfrac{C_i}{T_i^{min}} \\ x_i^{min} \longrightarrow U_i^{min} = \dfrac{C_i}{T_i^{max}} \\ k_i \longrightarrow \dfrac{1}{E_i} \end{cases}$$

The spring analogy

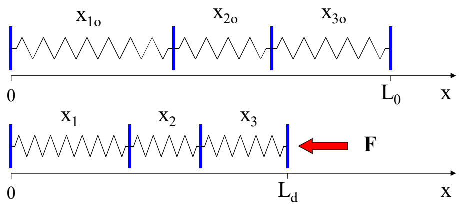
A periodic task set with maximum utilization U^{max} that must be reduced to a desired utilization U_d

can be treated as

a set of linear springs with initial length L_0 that must be compressed to reach a desired length L_d.



Solving a linear spring system



$$\begin{cases} F = k_1(x_{1o} - x_1) \\ F = k_2(x_{2o} - x_2) \\ F = k_3(x_{3o} - x_3) \end{cases} \qquad \begin{cases} x_1 + x_2 + x_3 = L_d \\ x_{1o} + x_{2o} + x_{3o} = L_0 \end{cases}$$

Solution assuming $x^{min} = 0$

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:

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

Solution assuming $x^{min} = 0$

Substituting F in the equations, we have:

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

That is:

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

Solution assuming $x^{min} = 0$

$$x_i = x_{io} - (L_0 - L_d)\frac{K_{//}}{k_i} \qquad K_{//} = \frac{1}{\sum_{i=1}^{n} \frac{1}{k_i}}$$

And defining: $E_i = 1/k_i$

$$x_i = x_{io} - (L_0 - L_d)\frac{E_i}{E_s} \qquad E_s = \sum_{i=1}^{n} E_i$$

Solution assuming $T^{max} = \infty$

Interpreting the solution for a task set we have:

$$x_i = x_{io} - (L_0 - L_d)\frac{E_i}{E_s}$$

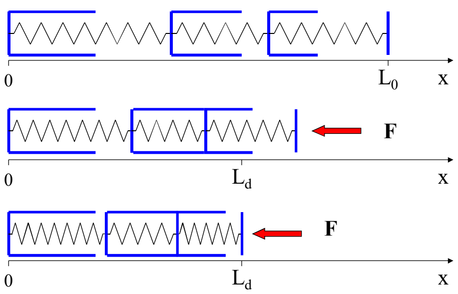
$$U_i = U_i^{max} - (U^{max} - U_d)\frac{E_i}{E_s}$$

Once the various U_i have been derived, task periods can be set as:

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

Solution with constraints

If $T^{max} < \infty$ (i.e., $x^{min} > 0$), the solution becomes iterative, requiring at most n iterations:



Solution with constraints

After each step, the set Γ can be divided into two subsets:

- a set Γ_f of fixed springs that reached the minimum length;
- a set Γ_v of variable springs that can still be compressed.

$$\forall \tau_i \in \Gamma_v \quad U_i = U_i^{max} - (U_v^{max} - U_d + U_f)\frac{E_i}{E_v}$$

$$U_v^{max} = \sum_{\tau_i \in \Gamma_v} U_i^{max} \qquad U_f = \sum_{\tau_i \in \Gamma_f} U_i^{min} \qquad E_v = \sum_{\tau_i \in \Gamma_v} E_i$$

If for some task $U_i < U_i^{min}$, then set $U_i = U_i^{min}$, update Γ_v and Γ_f and repeat the process.

Observations (1)

- **Feasibility condition**

  Given a task set with $U_{max} > U_d$, a compressed solution always exists if and only if $U^{min} \le U_d$.

- Initialization values of the iterative process:

$$\begin{cases} \Gamma_v = \Gamma \\ \Gamma_f = \{\} \end{cases} \qquad \begin{cases} U_v^{max} = U^{max} \\ U_f = 0 \\ E_v = E_s \end{cases}$$

- The computational complexity of the elastic compression algorithm is $O(n^2)$

Observations (2)

- The compression algorithm can be used to adjust periods every time a task is added to the system, or a task requests to adapt its period.
- The compression algorithm can also be used to increase utilizations when the overload is over or when a task set underutilize the processor.
- Elastic compression can also be used to compute how to reduce computation times ($C_i = U_i T_i$).