Course on
Real-Time Systems

Prof. Giorgio Buttazzo

Department of Computer Science
University of Pavia
E-mail: giorgio@sssup.it

Aim of the course

• Studing software technologies and methodologies for supporting complex computing systems.

• We will not consider how to control a system, but only how to provide a proper operating system support.
Course Outline - 1

- Basic concepts
- Where timing constraints come from?
- Task Scheduling
- Periodic task management
- Schedulability analysis
- Aperiodic server mechanisms
- Accessing shared resources

Course Outline - 2

- Predictable communication mechanisms
- Limits of the hard real-time approach
- Increasing flexibility
- Developing real-time applications
- HARTIK: a free real-time kernel
- Programming real-time applications
- Conclusions
General Definitions

In every control application, we can distinguish 3 basic components:

- **the system** to be controlled
  - it may include sensors and actuators
- **the controller**
  - it sends signals to the system according to a predetermined control objective
- **the environment** in which the system operates

A typical control system
### Detailed block diagram

![Block Diagram](image)

### Types of control systems

Depending on the system-environment interactions, we can distinguish 3 types of control systems:

- **Monitoring Systems**
  - do not modify the environment

- **Open-loop control systems**
  - loosely modify the environment

- **Closed-loop control systems**
  - tight interaction between perception and action
**Monitoring Systems**

Do not modify the environment

Examples: surveillance systems, air traffic control

---

**Open-loop control systems**

Sensing and control are loosely coupled

Examples: assembly robots, sorting robots
Closed-loop control systems

Sensing and control are tightly coupled

**Examples:** humans, flight control systems, military systems
**Implications**

- The tight interaction with the environment requires the system to react to events within precise timing constraints.
- Timing constraints are imposed by the dynamics of the environment.

> The operating system must be able to execute tasks within timing constraints.

**Real-time systems**

A computing system able to respond to events within precise timing constraints is called a **Real-Time System**.
What’s a real-time system?

It is a system in which the correctness depends not only on the output values, but also on the time at which results are produced.

REAL TIME means that system time must be synchronized with the time in the environment.
Typical RT applications

- military systems for defense
- control of chemical/nuclear power plants
- robotics
- railway switching systems
- automotive applications
- monitoring systems (air traffic control)
- telecommunication systems

Traditional Approach

- In spite of this large application domain, most of RT applications are designed using empirical techniques:
  - assembly programming
  - timing through dedicated timers
  - control through driver programming
  - priority manipulations
Disadvantages

- Tedious programming which heavily depends on the programmer’s ability
- Difficult code understanding
- Difficult software maintainability
- Difficult verification of timing constraints


Lessons learned

- Tests, although necessary, allow only a partial verification of system’s behavior.
- Predictability must be improved at the kernel level.
- Overload handling and fault-tolerance.
- Critical systems must be designed by making pessimistic assumptions.
Real-Time ≠ Fast

- A real-time system is **not** a fast system.
- Speed is always relative to a specific environment.
- Running faster is good, but does not guarantee a correct behavior.

Speed vs. Predictability

- The objective of a real-time system is to guarantee the timing behavior of each individual task.
- The objective of a fast system is to minimize the average response time of a task set. But …
  
  Don’t trust average when you have to guarantee individual performance
Sources of non determinism

- **Architecture**
  - cache, pipelining, interrupts, DMA

- **Operating system**
  - scheduling, synchronization, communication

- **Language**
  - lack of explicit support for time

- **Design methodologies**
  - lack of analysis and verification techniques

Definitions

- **Process** (or task)
  is a sequence of instructions that in the absence of other activities is continuously executed by the processor until completion.
Task states

A task is said to be:
- **ACTIVE**: if it can be executed by the CPU;
- **BLOCKED**: if it is waiting for an event;

An **active** task can be:
- **RUNNING**: if it is being executed by the CPU;
- **READY**: if it is waiting for the CPU.

Ready Queue

- The **ready** tasks are kept in a waiting queue, called the **ready queue**;

- The strategy for choosing the ready task to be executed on the CPU is the **scheduling algorithm**.
Scheduling

- A scheduling algorithm is said to be:
  
  - **preemptive**: if the running task can be temporarily suspended in the ready queue to execute a more important task.
  
  - **non preemptive**: if the running task cannot be suspended until completion.

Task State Transitions
Schedule

A schedule is a particular assignment of tasks to the processor.

Given a task set $\Gamma = \{\tau_1, \ldots, \tau_n\}$, a schedule is a mapping $\sigma : \mathbb{R}^+ \rightarrow \mathbb{N}$ such that $\forall t \in \mathbb{R}^+$, $\exists t_1, t_2$:

$$t \in [t_1, t_2) \quad \land \quad \forall t' \in [t_1, t_2) : \sigma(t) = \sigma(t')$$

$$\sigma(t) = \begin{cases} 
  k > 0 & \text{if } \tau_k \text{ is running} \\
  0 & \text{if the processor is idle}
\end{cases}$$

A sample schedule

At time $t_1$, $t_2$, $t_3$, and $t_4$ a context switch is performed.

Each interval $[t_i, t_{i+1})$ is called a time slice.
A preemptive schedule

Real-Time tasks

- $r_i$: request time (arrival time $a_i$)
- $s_i$: start time
- $C_i$: worst-case execution time (wcet)
- $d_i$: absolute deadline
- $D_i$: relative deadline
- $f_i$: finishing time
Other parameters

![Diagram of task criticality parameters]

- **Lateness**: \( L_i = f_i - d_i \)
- **Tardiness**: \( \max(0, L_i) \)
- **Residual wcet**: \( c_i(t) \quad c_i(r_i) = C_i \)
- **Laxity (or slack)**: \( d_i - t - c_i(t) \)

Task Criticality

**HARD tasks**

Missing a deadline may cause catastrophic effects on the controlled system.

**SOFT tasks**

Missing a deadline only causes a performance degradation.

An operating system able to handle hard tasks is called a **hard real-time** system.
Typical HARD tasks
– sensory acquisition
– low-level control
– sensory-motor planning

Typical SOFT tasks
– reading data from the keyboard
– user command interpretation
– message displaying
– graphical activities

Activation modes

- **Time driven**: periodic tasks
  the task is automatically activated by the kernel at regular intervals.

- **Event driven**: aperiodic tasks
  the task is activated upon the arrival of an event or through an explicit invocation of the activation primitive.
**Periodic task model**

\[
\begin{aligned}
    r_{i1} &= \Phi_i \\
    r_{i,k+1} &= r_{i,k} + T_i \\
\end{aligned}
\]

\[\tau_i(C_i, T_i, D_i)\]

\[
\begin{aligned}
    r_{i,k} &= \Phi_i + (k-1) T_i \\
    d_{i,k} &= r_{i,k} + D_i \\
    \text{often} \\
    D_i &= T_i
\end{aligned}
\]

**Aperiodic task model**

- **Aperiodic:** \( r_{i,k+1} > r_{i,k} \)
- **Sporadic:** \( r_{i,k+1} \geq r_{i,k} + T_i \)
Types of constraints

- **Timing constraints**
  - activation, completion, jitter.

- **Precedence constraints**
  - they impose an ordering in the execution.

- **Resource constraints**
  - they enforce a synchronization in the access of mutually exclusive resources.

Timing constraints

Can be **explicit** or **implicit**.

- **Explicit constraints**
  - Are included in the specification of the system activities.

**Examples**
- open the valve **in** 10 seconds
- send the position **within** 40 ms
- read the altimeter **every** 200 ms
• **Implicit constraints**
  – do not appear in the system specification, but must be respected to meet the requirements.

**Examples**
– Avoid obstacles while running at speed $v$.
– Slide a surface with speed $v$ and force $F$.

---

**Example: automatic breaking**

![Diagram showing automatic breaking system]

Human controls dashboard, which triggers the distribution unit. Sensors detect obstacles and condition checkers signal an emergency stop if necessary.
Worst-case reasoning

D = sensor visibility

\[ v(T_s + \Delta) + X_b < D \]

\[ \begin{align*}
X_b &= vt - \frac{1}{2}at^2 \\
v &= at
\end{align*} \]

\[ a = \mu g \]

\[ X_b = \frac{v^2}{2\mu g} \]

\[ v(T_s + \Delta) + \frac{v^2}{2\mu g} < D \]
\[ T_s < \frac{D}{v} - \frac{v}{2\mu g} - \Delta \]

\[ v_{\text{max}} \approx \sqrt{(\Delta \mu g)^2 + 2D\mu g - \Delta \mu g} \]

Lessons learned

\[ v_{\text{max}} \approx \sqrt{2D\mu g} \]

The farther we look, the faster we can run
To go fast safely, look ahead!!

If \( v \geq v_{\text{max}} \) no feasible solution exists, no matter how fast the processor is!!!
**Precedence constraints**

Sometimes tasks must be executed with specific precedence relations, specified by a **Directed Acyclic Graph**:

\[ \tau_1 \rightarrow \tau_2 \]
\[ \tau_1 \prec \tau_4 \]

**Sample application**

*stereo vision*

processing \[\rightarrow\] recognition
Resource constraints

To preserve data consistency, shared resources must be accessed in mutual exclusion:

\[
\begin{align*}
\tau_W & \quad x = 1 \\
& \quad y = 8 \\
\tau_R & \quad x = 3 \\
& \quad y = 5 \\
\end{align*}
\]
Mutual exclusion

However, mutual exclusion introduces extra delays:

Scheduling anomalies

priority

P_i > P_j \ \forall \ i < j

t_r = 12
Increased processors

$T_1$: 3  $T_9$: 9

$T_2$: 2  $T_8$: 4
$T_3$: 2  $T_7$: 4
$T_4$: 2  $T_6$: 4

Shorter tasks

$T_1$: 2  $T_9$: 8

$T_2$: 1  $T_8$: 3
$T_3$: 1  $T_7$: 3
$T_4$: 1  $T_6$: 3

$t_r = 15$

$t_r = 13$
Released constraints

\[ t_r = 16 \]

Faster processor

double speed

deadline miss
A dangerous operation: DELAY

A delay($\Delta$) may cause a delay longer than $\Delta$.

Lessons learned

- Tests are not enough for real-time systems
- Intuitive solutions do not always work
- Delay should not be used in real-time tasks

The safest approach:

- use predictable kernel mechanisms
- analyze the system to predict its behavior
Achieving predictability

- The operating system is the part most responsible for a predictable behavior.
- Concurrency control must be enforced by:
  - appropriate scheduling algorithms
  - appropriate synchronization protocols
  - efficient communication mechanisms
  - predictable interrupt handling

Let’s go studying real-time scheduling
Task scheduling

Definitions

- A schedule $\sigma$ is said to be **feasible** if all the tasks are able to complete within a set of constraints.

- A set of tasks $\Gamma$ is said to be **schedulable** if there exists a feasible schedule for it.
The general scheduling problem

Given a set $\Gamma$ of $n$ tasks, a set $P$ of $m$ processors, and a set $R$ of $r$ resources, find an assignment of $P$ and $R$ to $\Gamma$ which produces a feasible schedule.

Complexity

- In 1975, Garey and Johnson showed that the general scheduling problem is NP hard.
- However, polynomial time algorithms can be found under particular conditions.
Simplifying assumptions

- Single processor
- Homogeneous task sets
- Fully preemptive tasks
- Simultaneous activations
- No precedence constraints
- No resource constraints

Algorithm taxonomy

- Preemptive vs. Non Preemptive
- Static vs. dynamic
- On line vs. Off line
- Best Effort vs. Optimal
Static vs. Dynamic

**Static**

scheduling decisions are taken based on fixed parameters, statically assigned to tasks before activation.

**Dynamic**

scheduling decisions are taken based on parameters that can change with time.

Off-line vs. On-line

**Off-line**

all scheduling decisions are taken before task activation: the schedule is stored in a table.

**On-line**

scheduling decisions are taken at run time on the set of active tasks.
Best-Effort vs. Optimal

Best-Effort

do their best to find a feasible schedule, if there exists one, but they do not guarantee that.

Optimal

Always find a feasible schedule if there exists one.

Classical scheduling policies

- First Come First Served
- Shortest Job First
- Priority Scheduling
- Round Robin

*Not suited for real-time systems*
First Come First Served

It assigns the CPU to tasks based on their arrival times.

- Non preemptive
- Dynamic
- On line
- Best effort

First Come First Served

- Very unpredictable
  
  response times strongly depend on task arrivals.

\[
\begin{align*}
\tau_1 & = 20 \\
\tau_2 & = 26 \\
\tau_3 & = 26
\end{align*}
\]

\[
\begin{align*}
\tau_1 & = 26 \\
\tau_2 & = 8 \\
\tau_3 & = 2
\end{align*}
\]
Shortest Job First (SJF)

It selects the task with the shortest computation time.

- **Non preemptive** or **preemptive**
- **Static** ($C_i$ is a constant parameter)
- It can be used **on line** or **off-line**
- It minimizes the average response time

**SJF Optimality**

\[
R(\sigma') = \frac{1}{n} \sum_{i=1}^{n} (f'_{i} - r_{i}) \leq \frac{1}{n} \sum_{i=1}^{n} (f_{i} - r_{i}) = R(\sigma)
\]
**SJF Optimality**

\[ \sigma \rightarrow \sigma' \rightarrow \sigma'' \rightarrow \ldots \rightarrow \sigma^* \]

\[ \bar{R}(\sigma) \geq \bar{R}(\sigma') \geq \bar{R}(\sigma'') \ldots \geq \bar{R}(\sigma^*) \]

\[ \sigma^* = \sigma_{SJF} \]

\[ \bar{R}(\sigma_{SJF}) \] is the minimum response time achievable by any algorithm

---

**SJF suited for Real-Time?**

- It is not optimal in the sense of feasibility
Priority Scheduling

- Each task is assigned a priority: $p_i \in [0, 255]$
- The task with the highest priority is selected for execution.
- Tasks with the same priority are served FCFS.

  - Preemptive
  - Static or dynamic
  - On line

Priority Scheduling

- Problem: starving
  low priority tasks may experience long delays due to the preemption of high priority tasks.

- A solution: aging
  priority increases with waiting time

NOTE:

<table>
<thead>
<tr>
<th>Equation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_i \propto 1/C_i$</td>
<td>SJF</td>
</tr>
<tr>
<td>$p_i \propto 1/r_i$</td>
<td>FCFS</td>
</tr>
</tbody>
</table>
Round Robin

- The ready queue is served as FCFS, but ...
- Each task $\tau_i$ cannot execute more than $Q$ time units ($Q =$ time quantum).
- When $Q$ expires, $\tau_i$ is put back in the queue.

![Diagram of Round Robin](image_url)

Round Robin

$n =$ number of tasks in the system

$R_i \equiv (nQ) \frac{C_i}{Q} = nC_i$

**Time sharing**

Each task runs as it was executing alone on a virtual processor $n$ times slower than the real one.
Round Robin

- if \( Q > \max(C_i) \) then \( RR \equiv FCFS \)
- if \( Q \equiv \) context switch time (\( \delta \)) then

\[
R_i \equiv n(Q + \delta) \frac{C_i}{Q} = nC_i \left( \frac{Q + \delta}{Q} \right)
\]

Real-Time Algorithms

Tasks can be scheduled by

- relative deadlines \( D_i \) (static)
- absolute deadlines \( d_i \) (dynamic)
Earliest Due Date

It selects the task with the earliest relative deadline [Jackson’ 55].

- All tasks arrive simultaneously
- Fixed priority  (\(D_i\) is known in advance)
- Preemption  is not an issue
- It minimizes the maximum lateness (\(L_{\text{max}}\))

Lateness

\[ L_i = f_i - d_i \]

\(L_i > 0\)

\(L_i < 0\)
**Maximum Lateness**

\[ L_{\text{max}} = \max_i (L_i) \]

If \( L_{\text{max}} < 0 \) then no task misses its deadline.

**EDD Optimality**

\[ L_{\text{max}} = L_a = f_a - d_a \]
\[ L_{a'} = f_{a'} - d_a < f_a - d_a \]
\[ L_{b'} = f_{b'} - d_b < f_a - d_a \]
\[ L'_{\text{max}} < L_{\text{max}} \]
EDD Optimality

\[ \sigma \rightarrow \sigma' \rightarrow \sigma'' \rightarrow \ldots \rightarrow \sigma^* \]

\[ L_{\text{max}}(\sigma) \geq L_{\text{max}}(\sigma') \geq L_{\text{max}}(\sigma'') \ldots \geq L_{\text{max}}(\sigma^*) \]

\[ \sigma^* = \sigma_{\text{EDD}} \]

\[ L_{\text{max}}(\sigma_{\text{S/F}}) \] is the minimum value achievable by any algorithm

EDD - Guarantee test (off line)

A task set \( \Gamma \) is feasible if \( \forall i \ f_i \leq d_i \)

\[ f_i = \sum_{k=1}^{i} C_k \]

\[ \forall i \ \sum_{k=1}^{i} C_k \leq D_i \]
Earliest Deadline First

It selects the task with the earliest **absolute** deadline [Horn 74].

- Tasks may arrive at any time
- Dynamic priority \( (d_i \text{ depends on arrival}) \)
- Full preemptive tasks
- It minimizes the maximum lateness \( (L_{\text{max}}) \)

EDF Example
EDF Guarantee test (on line)

\[ \forall i \sum_{k=1}^{i} c_k(t) \leq d_i - t \]

Complexity Issues

**EDD**

- \(O(n \log n)\) to order the task set
- \(O(n)\) to guarantee the whole task set

**EDF**

- \(O(n)\) to insert a new task in the queue
- \(O(n)\) to guarantee a new task
Periodic Task Scheduling

Problem formulation

For each periodic task, guarantee that:

- each job $\tau_{ik}$ is activated at $r_{ik} = (k-1)T_i$
- each job $\tau_{ik}$ completes within $d_{ik} = r_{ik} + D_i$
Timeline Scheduling (cyclic scheduling)

It has been used for 30 years in military systems, navigation, and monitoring systems.

Examples

– Air traffic control
– Space Shuttle
– Boeing 777

Timeline Scheduling

Method

• The time axis is divided in intervals of equal length (*time slots*).

• Each task is statically allocated in a slot in order to meet the desired request rate.

• The execution in each slot is activated by a timer.
Example

<table>
<thead>
<tr>
<th>task</th>
<th>f</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>40 Hz</td>
<td>25 ms</td>
</tr>
<tr>
<td>B</td>
<td>20 Hz</td>
<td>50 ms</td>
</tr>
<tr>
<td>C</td>
<td>10 Hz</td>
<td>100 ms</td>
</tr>
</tbody>
</table>

\[ \Delta = \text{MCD (minor cycle)} \]
\[ T = \text{mcm (major cycle)} \]

Guarantee:

\[ C_A + C_B \leq \Delta \]
\[ C_A + C_C \leq \Delta \]

Implementation

\[ \text{minor cycle} \]
\[ \text{major cycle} \]
Timeline scheduling

Advantages

• Simple implementation (no real-time operating system is required).
• Low run-time overhead.
• It allows jitter control.

Disadvantages

• It is not robust during overloads.
• It is difficult to expand the schedule.
• It is not easy to handle aperiodic activities.
Problems during overloads

What do we do during task overruns?

• Let the task continue
  – we can have a domino effect on all the other tasks (timeline break)

• Abort the task
  – the system can remain in inconsistent states.

Expandibility

If one or more tasks need to be upgraded, we may have to re-design the whole schedule again.

Example: B is updated but $C_A + C_B > \Delta$
Expandibility

- We have to split task B in two subtasks (B₁, B₂) and re-build the schedule:

![Diagram showing task scheduling]

**Guarantee:**
\[
\begin{align*}
C_A + C_{B1} & \leq \Delta \\
C_A + C_{B2} + C_C & \leq \Delta
\end{align*}
\]

Expandibility

If the frequency of some task is changed, the impact can be even more significant:

<table>
<thead>
<tr>
<th>Task</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>25 ms</td>
<td>25 ms</td>
</tr>
<tr>
<td>B</td>
<td>50 ms</td>
<td>40 ms</td>
</tr>
<tr>
<td>C</td>
<td>100 ms</td>
<td>100 ms</td>
</tr>
</tbody>
</table>

**Minor cycle:** \( \Delta = 25 \) \( \Delta = 5 \)

**Major cycle:** \( T = 100 \) \( T = 200 \) \( 40 \) sync. per cycle!
Priority Scheduling

Method

• Each task is assigned a priority based on its timing constraints.
• We verify the feasibility of the schedule using analytical techniques.
• Tasks are executed on a priority-based kernel.
Rate Monotonic (RM)

- Each task is assigned a fixed priority proportional to its rate [Liu & Layland ‘73].

![Task Schedules]

How can we verify feasibility?

- Each task uses the processor for a fraction of time:
  \[ U_i = \frac{C_i}{T_i} \]
- Hence the total processor utilization is:
  \[ U_p = \sum_{i=1}^{n} \frac{C_i}{T_i} \]
- \( U_p \) is a measure of the processor load
A necessary condition

If $U_p > 1$ the processor is overloaded hence the task set cannot be schedulable.

However, there are cases in which $U_p < 1$ but the task is not schedulable by RM.

An unfeasible RM schedule

$$U_p = \frac{3}{6} + \frac{4}{9} = 0.944$$

deadline miss
Utilization upper bound

\[ U_p = \frac{3}{6} + \frac{3}{9} = 0.833 \]

**NOTE:** If \( C_1 \) or \( C_2 \) is increased, \( \tau_2 \) will miss its deadline!

A different upper bound

\[ U_p = \frac{2}{4} + \frac{4}{8} = 1 \]

The upper bound \( U_{ub} \) depends on the specific task set.
The least upper bound

\[ U_{ub} \]

\[ 1 \]

\[ U_{ub} \]

\[ \Gamma \]

A sufficient condition

If \( U_p \leq U_{ub} \) the task set is certainly schedulable with the RM algorithm.

NOTE

If \( U_{ub} < U_p \leq 1 \) we cannot say anything about the feasibility of that task set.
In 1973, Liu and Layland proved that for a set of $n$ periodic tasks:

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

for $n \to \infty$, $U_{lub} \to \ln 2$
RM Guarantee Test

• We compute the processor utilization as:

\[ U_p = \sum_{i=1}^{n} \frac{C_i}{T_i} \]

• Guarantee Test (only sufficient):

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

Basic Assumptions

A1. \( C_i \) is constant for every instance of \( \tau_i \)

A2. \( T_i \) is constant for every instance of \( \tau_i \)

A3. For each task, \( D_i = T_i \)

A4. Tasks are independent:
  • no precedence relations
  • no resource constraints
RM Optimality

RM is optimal among all fixed priority algorithms:

If there exists a fixed priority assignment which leads to a feasible schedule for $\Gamma$, then the RM assignment is feasible for $\Gamma$.

If $\Gamma$ is not schedulable by RM, then it cannot be scheduled by any fixed priority assignment.

Critical Instant

For any task $\tau_i$, the longest response time occurs when it arrives together with all higher priority tasks.
Computing $U_{ub}$

- Assume the worst-case scenario for the task set (simultaneous arrivals)
- Fully utilize the processor
- Compute the upper bound $U_{ub}$
- Minimize $U_{ub}$ with respect to all remaining variables

Computing $U_{ub}$ for 2 tasks

\[
C_{2\text{ max}} = T_2 - (F+1) C_1
\]
\[
F = \left[ \frac{T_2}{T_1} \right]
\]
\[
U_{ub} = \frac{C_1}{T_1} + \frac{T_2 - (F+1)C_1}{T_2} = 1 + \frac{C_1}{T_2} \left[ \frac{T_2}{T_1} - (F+1) \right]
\]
Computing $U_{lb}$ for 2 tasks

$$U_{lb} = 1 + \frac{C_1}{T_2} T_2 - (F + 1)$$

And minimizing with respect to $T_2/T_1$ we have:

$$U_{lb} = 2\left(\sqrt{2} - 1\right) \approx 0.83$$
Earliest Deadline First (EDF)

• Each job receives an absolute deadline:
  \[ d_{i,k} = r_{i,k} + D_i \]

• At any time, the processor is assigned to the job with the earliest absolute deadline.

• Under EDF, any task set can utilize the processor up to 100%.

EDF Example

\[ U_p = \frac{3}{6} + \frac{4}{9} = 0.94 \]

\[ D_i = T_i \]
The RM unfeasible schedule

\[ U_p = \frac{3}{6} + \frac{4}{9} = 0.944 \]

EDF Optimality

EDF is **optimal** among all algorithms:

If there exists a feasible schedule for \( \Gamma \), then EDF will generate a feasible schedule.

If \( \Gamma \) is not schedulable by EDF, then it cannot be scheduled by any algorithm.
EDF Optimality

[Dertouzos '74]

\[
\sigma \begin{cases}
\tau_E \\
\tau_k
\end{cases}
\]

Transforming \( \sigma \) in \( \sigma' \)

\[
\begin{align*}
\sigma'(t) &= \sigma(t_E) \\
\sigma'(t_E) &= \sigma(t)
\end{align*}
\]

Feasibility is preserved

\[
f'_k = f_E \leq d_E \leq d_k
\]

EDF schedulability

- In 1973, Liu and Layland proved that for a set of \( n \) periodic tasks:

\[
U_{lub}^{EDF} = 1
\]

- This means that a task set \( \Gamma \) is schedulable by EDF if and only if

\[
U_p \leq 1
\]
Proving sufficiency

\[ U_p \leq 1 \quad \Rightarrow \quad \Gamma \text{ schedulable} \]

- We find any algorithm for which the above condition holds;
- Then, for the EDF optimality, we can say that the above condition also holds for EDF.

Consider the algorithm which schedules in every interval of length \( \Delta \) a fraction of task:

\[ \delta_i = U_i \Delta \]
Proving sufficiency

With this algorithm, a task executes in each period for:

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

Feasibility is ensured if \( \sum_{i=1}^{n} \delta_i \leq \Delta \) that is if

\[ \sum_{i=1}^{n} U_i \Delta \leq \Delta \rightarrow U_p \leq 1 \]

RM vs. EDF

**RM**

- \( \tau_1 \)
- \( \tau_2 \)

**EDF**

- \( \tau_1 \)
- \( \tau_2 \)

deadline miss
RM vs. EDF

EDF

• It’s more efficient
• It reduces context switches

RM

• It is simpler to implement on commercial operating systems
• More predictable during overloads

Extension to tasks with $D < T$

Scheduling algorithms

• Deadline Monotonic: $p_i \propto 1/D_i$ (static)
• Earliest Deadline First: $p_i \propto 1/d_i$ (dynamic)
Deadline Monotonic

![Deadline Monotonic Chart]

Problem with the Utilization Bound

\[ U_p = \sum_{i=1}^{n} \frac{C_i}{D_i} = \frac{2}{3} + \frac{3}{6} = 1.16 > 1 \]

but the task set is schedulable.

Response Time Analysis

[Audsley '90]

- For each task \( \tau_i \), compute the interference due to higher priority tasks:
  \[ I_i = \sum_{D_k < D_i} C_k \]

- Compute its response time as
  \[ R_i = C_i + I_i \]

- Verify if \( R_i \leq D_i \)
Computing Interference

Interference of $\tau_k$ on $\tau_i$ in the interval $[0, R_i]$: 

$$I_{ik} = \left\lceil \frac{R_i}{T_k} \right\rceil C_k$$

Interference of high priority tasks on $\tau_i$: 

$$I_i = \sum_{k=1}^{i-1} \left\lceil \frac{R_i}{T_k} \right\rceil C_k$$

Computing Response Time

$$R_i = C_i + \sum_{k=1}^{i-1} \left\lceil \frac{R_{(s-1)}^i}{T_k} \right\rceil C_k$$

Iterative solution:

\[
\begin{align*}
R_i^0 &= C_i \\
R_i^s &= C_i + \sum_{k=1}^{i-1} \left\lceil \frac{R_{(s-1)}^i}{T_k} \right\rceil C_k
\end{align*}
\]

iterate until $R_i^s > R_i^{(s-1)}$
Dynamic Priority

EDF
Schedule based on absolute deadlines

Schedulability Analysis

Processor Demand Criterion [Baruah ‘90]

In any interval, the computation demanded by the task set must be no greater than the available time.

Processor Demand

The demand in \([t_1, t_2]\) is the computation time of those tasks started at or after \(t_1\) with deadline less than or equal to \(t_2\):

\[
g(t_1, t_2) = \sum_{t_i \geq t_1} \sum_{d_i \leq t_2} C_i
\]
Processor Demand

\[ g(0, L) = \sum_{i=1}^{n} \left[ \frac{L - D_i + T_i}{T_i} \right] C_i \]

Processor Demand in \([0, L]\)

\[ \forall L > 0, \quad g(0, L) \leq L \]

Question

How can we bound the number of intervals in which the test has to be performed?
Bounding complexity

- Since $g(0,L)$ is a step function, we can check feasibility only at deadline points.

- If tasks are synchronous and $U_p < 1$, we can check feasibility up to the hyperperiod $H$:

$$H = \text{lcm}(T_1, \ldots, T_n)$$
Bounding complexity

• Moreover we note that: \( g(0, L) \leq G(0, L) \)

\[
G(0, L) = \sum_{i=1}^{n} \left( \frac{L + T_i - D_i}{T_i} \right) C_i
\]

\[
= \sum_{i=1}^{n} L \frac{C_i}{T_i} + \sum_{i=1}^{n} (T_i - D_i) \frac{C_i}{T_i}
\]

\[
= LU + \sum_{i=1}^{n} (T_i - D_i) U_i
\]

Limiting L

\[
G(0, L) = LU + \sum_{i=1}^{n} (T_i - D_i) U_i
\]

\[
L^* = \frac{\sum_{i=1}^{n} (T_i - D_i) U_i}{1-U}
\]

for \( L > L^* \)

\( g(0,L) \leq G(0,L) < L \)
Processor Demand Test

\[ \forall L \in D, \quad g(0, L) \leq L \]

\[ D = \{d_k \mid d_k \leq \min (H, L^*)\} \]

\[
\begin{align*}
H &= \text{lcm}(T_1, \ldots, T_n) \\
L^* &= \frac{\sum_{i=1}^{n} (T_i - D_i) U_i}{1 - U}
\end{align*}
\]

Summary

- **Three scheduling approaches:**
  - Off-line construction (Timeline)
  - Fixed priority (RM, DM)
  - Dynamic priority (EDF)

- **Three analysis techniques:**
  - Processor Utilization Bound \( U \leq U_{\text{lub}} \)
  - Response Time Analysis \( \forall i \quad R_i \leq D_i \)
  - Processor Demand Criterion \( \forall L \quad g(0, L) \leq L \)
Complexity Issues

- **Utilization based analysis** \( (U \leq U_{\text{ub}}) \)
  - \( O(n) \) complexity

- **Response time analysis** \( (\forall i \ R_i \leq D_i) \)
  - Pseudo-polynomial complexity

- **Processor demand analysis** \( (\forall L \ g(0,L) \leq L) \)
  - Pseudo-polynomial complexity
Handing Hybrid Task Sets

Periodic tasks
+
Aperiodic tasks

Aperiodic task handling

- Aperiodic tasks are typically activated by external events (given by interrupts).
- From one hand, we want to reduce the response times of aperiodic tasks.
- On the other hand, we don’t want to jeopardize schedulability of periodic tasks.
Handling Criticality

- Aperiodic tasks with **HARD** deadlines must be guaranteed under worst-case conditions.
- Off-line guarantee is only possible if we can bound interarrival times (**sporadic tasks**).
- Hence **sporadic tasks** can be guaranteed as periodic tasks with \( C_i = WCET_i \) and \( T_i = MIT_i \)

\[
\begin{align*}
WCET & = \text{Worst-Case Execution Time} \\
MIT & = \text{Minimum Interarrival Time}
\end{align*}
\]

SOFT aperiodic tasks

- Aperiodic tasks with **SOFT** deadlines should be executed as soon as possible, but without jeopardizing HARD tasks.
- We may be interested in
  - minimizing the average response time
  - performing an on-line guarantee
Periodic Scheduling
(EDF)

\[ \tau_1 \]
\[ \tau_2 \]
ape

\[ C_1 = 1 \]
\[ C_2 = 3 \]

Immediate service

\[ \tau_1 \]
\[ \tau_2 \]
ape

\[ C_1 = 1 \]
\[ C_2 = 3 \]

deadline miss

Response Time = 3
Background service

\[ \tau_1 \quad C_s = 1 \]

\[ \tau_2 \quad C_s = 3 \]

ape

0 2 4 6 8 10 12

Response Time = 10

Aperiodic Servers

- A server is a kernel activity aimed at controlling the execution of aperiodic tasks.
- Normally, a server is a periodic task having two parameters:
  \[ \begin{align*}
  C_s & \quad \text{capacity (or budget)} \\
  T_s & \quad \text{server period}
  \end{align*} \]

To preserve periodic tasks, no more than \( C_s \) units must be executed every period \( T_s \).
Aperiodic service queue

- The server is scheduled as any periodic task.
- Priority ties are broken in favor of the server.
- Aperiodic tasks can be selected using an arbitrary queueing discipline.

Fixed-priority Servers

- Polling Server
- Deferrable Server
- Sporadic Server
- Slack Stealer
Dynamic-priority Servers

- Dynamic Polling Server
- Dynamic Deferrable Server
- Dynamic Sporadic Server
- Total Bandwidth Server
- Constant Bandwidth Server

Polling Server (PS)

- At the beginning of each period, the budget is recharged at its maximum value.
- Budget is consumed during job execution.
- When the server becomes active and there are no pending jobs, $C_s$ is discharged to zero.
- When the server becomes active and there are pending jobs, they are served until $C_s > 0$. 
**PS properties**

- In the worst-case, the PS behaves as a periodic task with utilization $U_s = C_s / T_s$.

- Aperiodic tasks execute at the highest priority if $T_s = \min(T_1, \ldots, T_n)$.

- Liu & Layland analysis gives that:

$$U_{\text{lub}}^{\text{RM+PS}}(n) = U_s + n \left[ \frac{2}{U_s + 1} \right]^{1/n} - 1$$
Deferrable Server (DS)

- Is similar to the PS, but the budget is not discharged if there are no pending requests.
- Keeping the budget improves responsiveness, but decreases the utilization bound.
- Liu & Layland analysis gives that:

\[
U_{\text{lub}}^{RM+PS}(n \to \infty) = \ln(2) + n \left( \frac{U_s + 2}{2U_s + 1} \right)^{\frac{1}{n}} - 1
\]
**RM + DS schedulability**

\[ U^{RM + DS}_{lab} (n \to \infty) \]

\[ \ln 2 \]

**RM + Deferrable Server**

\( \tau_1 \) \( C_1 = 2 \)

\( \tau_2 \) \( C_2 = 1 \)

ape

PS \( C_s = 1 \)

\( T_s = 5 \)

Response Time = 4
Total Bandwidth Server (TBS)

- It is a dynamic priority server, used along with EDF.

- Each aperiodic request is assigned a deadline so that the server demand does not exceed a given bandwidth $U_s$.

- Aperiodic jobs are inserted in the ready queue and scheduled together with the HARD tasks.

![The TBS mechanism]

- Deadlines ties are broken in favor of the server.
- Periodic tasks are guaranteed if and only if $U_p + U_s \leq 1$.
**Deadline assignment rule**

- If job $J_k$ with computation time $C_k$ arrives at time $r_k$, it is assigned a deadline:
  $$d_k = r_k + \frac{C_k}{U_s}$$

- To keep track of the bandwidth assigned to previous jobs, $d_k$ must be computed as:
  $$d_k = \max (r_k, d_{k-1}) + \frac{C_k}{U_s}$$

**EDF + TBS schedule**

- $U_s = 1 - U_p = 1/4$
- $d_1 = r_1 + C_1 / U_s = 1 + 2 \cdot 4 = 9$
- $d_2 = \max (r_2, d_1) + C_2 / U_s = 9 + 1 \cdot 4 = 13$
Problems with the TBS

- Without a budget management, there is no protection against execution overruns.
- If a job executes more than expected, hard tasks could miss their deadlines.

\[ \tau_i \quad C_i = 1 \quad \text{overrun} \quad \text{deadline miss} \]

\[ U_s = 1/4 \]

Solution: task isolation

- In the presence of overruns, only the faulty task should be delayed.
- Each task \( \tau_i \) should not demand more than its declared utilization \( (U_i = C_i/T_i) \).
- If a task executes more than expected, its priority should be decreased (i.e., its deadline postponed).
Bandwidth partitioning

- Ideally, each task should be assigned a given bandwidth and never demand more.

Questions

- **What do we do if a task overruns?**
  - Only that task should be delayed.

- **Consequences**
  - if the task is hard  =>  exception
  - if the task is soft  =>  QoS degradation
Achieving isolation

- Isolation among tasks can be achieved through a bandwidth reservation.
- Each task is managed by a dedicated server having bandwidth $U_s$.
- The server assigns priorities (or deadlines) to tasks so that they do not exceed the reserved bandwidth.

Implementation

- $\tau_1$, $\tau_2$, $\tau_3$ are tasks.
- Each task is assigned to a server.
- Bandwidths assigned to servers are $U_{s1}$, $U_{s2}$, and $U_{s3}$.
- Tasks are dispatched to the CPU through an EDF ready queue.
- The condition $U_{s1} + U_{s2} + U_{s3} \leq 1$ ensures that the total bandwidth does not exceed 1.
Constant Bandwidth Server (CBS)

- It assigns deadlines to tasks as the TBS, but keeps track of job executions through a budget mechanism.
- When the budget is exhausted it is immediately replenished, but the deadline is postponed to keep the demand constant.

CBS parameters

Given by the user
- Maximum budget: $Q_s$
- Server period: $T_s$

$U_s = Q_s / T_s$ (server bandwidth)

Maintained by the server
- Current budget: $c_s$ (initialized to 0)
- Server deadline: $d_s$ (initialized to 0)
Basic CBS rules

- Arrival of job $J_k \Rightarrow$ assign $d_s$
  
  $\begin{align*}
  \text{if } (r_k + c_s/U_s \leq d_s) \text{ then recycle } d_s \\
  \text{else } & \begin{cases} 
    d_s = r_k + T_s \\
    c_s = Q_s
  \end{cases}
  \end{align*}$

- Budget exhausted $\Rightarrow$ postpone $d_s$
  
  $\begin{align*}
  \begin{cases} 
    d_s = d_s + T_s \\
    c_s = Q_s
  \end{cases}
  \end{align*}$

Deadline assignment

- $Q_s = 6$
- $T_s = 12$
Budget exhausted

Qₜ = 3
Tₜ = 6

EDF + CBS schedule

CBS: Qₜ = 2, Tₜ = 6
CBS properties

• Bandwidth Isolation
  If a task \( \tau_i \) is served by a CBS with bandwidth \( U_s \) then, in any interval \( \Delta t \), \( \tau_i \) will never demand more than \( U_s \Delta t \).

• Hard schedulability
  A hard task \( \tau_i (C_i, T_i) \) is schedulable by a CBS with \( Q_s = C_i \) and \( T_s = T_i \), iff \( \tau_i \) is schedulable by EDF.

Remarks on the CBS

• It can be used as a safe server for handling aperiodic tasks under EDF.

• It can be used as a bandwidth reservation mechanism to achieve task isolation.

• It allows to guarantee a minimum performance to SOFT tasks, based on the assigned bandwidth.
Handling shared resources

Problems caused by mutual exclusion

Critical sections

$\tau_1$
- wait(s)
- $x = 3$
- $y = 5$
- signal(s)

write

int $x$
int $y$

read

global memory buffer

$\tau_2$
- wait(s)
- $a = x+1$
- $b = y+2$
- $c = x+y$
- signal(s)
Blocking on a semaphore

It seems that the maximum blocking time for $\tau_1$ is equal to the length of the critical section of $\tau_2$, but …

Schedule with no conflicts
Conflict on a critical section

priority

BCT

SCT

MT

Conflict on a critical section

priority

BCT

SCT

MT
Priority Inversion

A high priority task is blocked by a lower-priority task for an unbounded interval of time.

Solution

Introduce a concurrency control protocol for accessing critical sections.

Resource Access Protocols

- Non Preemptive Protocol (NPP)
- Highest Locker Priority (HLP)
- Priority Inheritance Protocol (PIP)
- Priority Ceiling Protocol (PCP)
- Stack Resource Policy (SRP)
Non Preemptive Protocol

- Preemption is forbidden in critical sections.
- Implementation: when a task enters a CS, its priority is increased at the maximum value.

ADVANTAGES: simplicity

PROBLEMS: high priority tasks that do not use the CS may also block

Conflicts on critical section

<table>
<thead>
<tr>
<th>Priority</th>
<th>τ₁</th>
<th>τ₂</th>
<th>τ₃</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

B
Schedule with NPP

Priority

$\tau_1$

$\tau_2$

$\tau_3$

$P_{CS} = \max\{P_1, \ldots, P_n\}$

Problem with NPP

Priority

useless blocking

$\tau_1$

$\tau_2$

$\tau_3$

$\tau_1$ cannot preempt, although it could
Highest Locker Priority

A task in a CS gets the highest priority among the tasks that use it.

FEATURES:

- Simple implementation.
- A task is blocked when attempting to preempt, not when entering the CS.

Schedule with HLP

\[
P_{CS} = \max \{ P_k | \tau_k \text{ uses CS} \}
\]

\( \tau_2 \) is blocked, but \( \tau_1 \) can preempt within a CS
Problem with HLP

\[ \tau_1 \] blocks just in case ...

Priority Inheritance Protocol
[Sha, Rajkumar, Lehoczky, 90]

- A task in a CS increases its priority only if it blocks other tasks.
- A task in a CS inherits the highest priority among those tasks it blocks.

\[ P_{CS} = \max \{ P_k \mid \tau_k \text{ blocked on CS} \} \]
Schedule with PIP

priority

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

\[ p_1 \]
\[ p_3 \]

direct blocking

push-through blocking

Types of blocking

- **Direct blocking**
  A task blocks on a locked semaphore

- **Push-through blocking**
  A task blocks because a lower priority task inherited a higher priority.

**BLOCKING:**
a delay caused by a lower priority task
Identifying blocking resources

- A task $\tau_i$ can be blocked by those semaphores used by lower priority tasks and
  - directly shared with $\tau_i$ (direct blocking) or
  - shared with tasks having priority higher than $\tau_i$ (push-through blocking).

**Theorem:** $\tau_i$ can be blocked at most once by each of such semaphores

Example

<table>
<thead>
<tr>
<th>Priority</th>
<th>$\tau_1$</th>
<th>$\tau_2$</th>
<th>$\tau_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td></td>
<td>D</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- $\tau_1$ can be blocked once by $\tau_2$ (on A or C) and once by $\tau_3$ (on B or D)
- $\tau_2$ can be blocked once by $\tau_3$ (on B or D)
- $\tau_3$ cannot be blocked
Bounding blocking times

- If $n$ is the number of tasks with priority less than $\tau_i$
- and $m$ is the number of semaphores on which $\tau_i$ can be blocked, then

**Theorem:** $\tau_i$ can be blocked at most for the duration of $\min(n,m)$ critical sections
Remarks on PIP

ADVANTAGES
- It is transparent to the programmer.
- It bounds priority inversion.

PROBLEMS
- It does not avoid deadlocks and chained blocking.

Chained blocking with PIP

Theorem: $\tau_1$ can be blocked at most once by each lower priority task.
**Priority Ceiling Protocol**

- Can be viewed as PIP + access test.
- A task can enter a CS only if it is free and there is no risk of chained blocking.

To prevent chained blocking, a task may stop at the entrance of a free CS (*ceiling blocking*).

**Resource Ceilings**

- Each semaphore $s_k$ is assigned a ceiling:

$$C(s_k) = \max \{P_j : \tau_j \text{ uses } s_k\}$$

- A task $\tau_i$ can enter a CS only if

$$P_i > \max \{C(s_k) : s_k \text{ locked by tasks } \neq \tau_i\}$$
Schedule with PCP

\[ s_1 \quad C(s_1) = P_1 \]
\[ s_2 \quad C(s_2) = P_1 \]

\[ t_1: \tau_2 \text{ is blocked by the PCP, since } P_2 < C(s_1) \]

Remarks on PCP

**ADVANTAGES**
- Blocking is reduced to only one CS
- It prevents deadlocks

**PROBLEMS**
- It is not transparent to the programmer: semaphores need ceilings
**Guarantee with resource constraints**

- We select a scheduling algorithm and a resource access protocol.
- We compute the maximum blocking times ($B_i$) for each task.
- We perform the guarantee test including the blocking terms.

**Guarantee with RM**

\[
\forall i \sum_{k=1}^{i-1} \frac{C_k}{T_k} + \frac{C_i + B_i}{T_i} \leq i(2^{1/i} - 1)
\]
Guarantee with EDF

\[ \forall i \sum_{k=1}^{i-1} \frac{C_k}{T_k} + \frac{C_i + B_i}{T_i} \leq 1 \]
Overload Management

Predictability vs. Efficiency

Causes of overloads

- Bad system design (based on average behavior)
- Malfunctioning of input devices
- Variations in the environment
- Simultaneous arrivals of events
- Exceptions raised by the kernel
Load definitions

• Soft aperiodic tasks: \( \rho = \lambda \cdot \bar{C} \)

• Hard periodic tasks: \( \rho = U = \sum_{i=1}^{n} \frac{C_i}{T_i} \)

• General RT tasks:
  if \( g(t_1, t_2) \) is the processor demand in \([t_1, t_2]\), then:
  \[ \rho = \max_{t_1, t_2} \frac{g(t_1, t_2)}{t_2 - t_1} \]

Istantaneous load \( \rho(t) \)

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

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

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

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

A matter of cost

- High predictability and low efficiency means wasting resources $\Rightarrow$ high cost
  - it can be justified only for critical systems

- High efficiency in resource usage implies:
  - ability to handle and tolerate overloads
  - graceful degradation
  - exception handling mechanisms
Overload management

Overload situations 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

Existing techniques

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

**Performance degradation**
- Imprecise Computation
- Job Skipping
- Adaptive 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:

<table>
<thead>
<tr>
<th>$v_i$</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_i = V_i$</td>
<td>arbitrary constant</td>
</tr>
<tr>
<td>$v_i = C_i$</td>
<td>computation time</td>
</tr>
<tr>
<td>$v_i = V_i/C_i$</td>
<td>value density</td>
</tr>
</tbody>
</table>

Value as a function of time

In a real-time system, the value of a task depends on its completion time and criticality:
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: $\Gamma_A(T) < \Gamma_{\text{max}}(T) = \sum_{i=1}^{n} V_i$

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 (assume: $V_i = C_i$)

To maximize $\Gamma_A$ we should know the future.

If at time $t = 0$ $r_j$ is not know, we cannot select the task that maximizes the cumulative value.
Competitive Factor

- Let $\Gamma^*$ 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 $\varphi_{\text{EDF}} = 0$:

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 $V_2 >> V_1$
A theoretical upper bound

[Baruah et al., 91]

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

That is: $\max_A \phi_A \leq 0.25$

In general, the upper bound of the competitive factor is a function of the load and varies as follows:

![Graph showing the relationship between $\rho$ and $\phi_{on}$]
Best-effort scheduling

- Tasks are always accepted in the system.
- Performance is controlled through a suitable (value-based) priority assignment.
- **Problem:** domino effect.

Admission control

- Every task is subject to an acceptance test which keeps the load $\leq 1$.
- It prevents domino effects, but does not take values into account.
- Low efficiency due to the worst-case guarantee (tasks may be unnecessarily rejected).
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.

Robust EDF

- **Scheduling Policy** ⇒ **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

at time $t = 4 \implies \tau_3$ rejected
**Performance Degradation**

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

This can be done by:

- reducing precision of results
- skipping some jobs;
- relaxing timing constraints.
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

Imprecise computation

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

- a **mandatory** part: $\tau_{mi} (M_i, D_i)$
- an **optional** part: $\tau_{oi} (O_i, D_i)$

$w_i$ 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**: \( \bar{\varepsilon} = \frac{1}{n} \sum_{i=1}^{n} w_i \varepsilon_i \)

**GOAL**: minimize the average error

---

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 number of jobs that must be executed 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**.

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

---

**Example**

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

![Example Diagram](image)
Equivalent utilization factor

\[ U^*_p = \max_{L \geq 0} \left\{ \sum_{i=1}^{n} \frac{g_i(0,L)}{L} \right\} \]

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

Schedulability Analysis

A sufficient condition

**Theorem**: A set of firm periodic tasks is schedulable 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.

Bandwidth saving

- In general, skipping jobs of periodic tasks causes a bandwidth saving:

\[ \Delta U = U_p - U_p^* \]

- Such a bandwidth can be used for
  - improving aperiodic responsiveness (by increasing their reserved bandwidth);
  - accepting a larger number of periodic tasks.
Not always skips save bandwidth:

In this case: \( U_p^* = 1 \)

In fact, for \( L = T_i \) we have \( g_i(0,L) = C_i = T_i \)

Hence: \( \frac{g_i(0,L)}{L} = \frac{T_i}{T_i} = 1 \)

However, notice that:

In this case we still have: \( U_p^* = 1 \)

In fact: \( g(0, T_1) = T_1 \) and \( g(0, T_2) = T_2 \)

Hence: \( \frac{g(0, T_1)}{T_1} = \frac{g(0, T_2)}{T_2} = 1 \)
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.

Example

<table>
<thead>
<tr>
<th>task</th>
<th>C_i</th>
<th>T_0</th>
<th>T_{min}</th>
<th>T_{max}</th>
</tr>
</thead>
<tbody>
<tr>
<td>τ_1</td>
<td>10</td>
<td>20</td>
<td>20</td>
<td>25</td>
</tr>
<tr>
<td>τ_2</td>
<td>10</td>
<td>40</td>
<td>40</td>
<td>50</td>
</tr>
<tr>
<td>τ_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 $\tau_4$ 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:

$$U_p = \frac{10}{23} + \frac{10}{50} + \frac{15}{80} + \frac{5}{30} = 0.99$$

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 $T_{min} = T_{max}$, is equivalent to a hard task.

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

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

The linear spring analogy

\[
\begin{align*}
F &= k_1(x_{1o} - x_1) \\
F &= k_2(x_{2o} - x_2) \\
F &= k_3(x_{3o} - x_3)
\end{align*}
\]
Solution without constraints

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:

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

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:

\[ x_1 = x_{1o} - (L_0 - L_d) \frac{1/k_1}{\frac{1}{k_1} + \frac{1}{k_2} + \frac{1}{k_3}} \]
Solution without constraints

\[ x_i = x_{io} - (L_0 - L_d) \frac{K_{ll}}{k_i} \]

And defining: \( E_i = 1/k_i \)

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

\[ E_s = \sum_{i=1}^{n} E_i \]

Period computation

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

And then:

\[ T_i = \frac{C_i}{U_i} \]
Solution with constraints

Iterative solution:

Other use of elastic tasks

- Increase frequencies to fully utilize the processor.
- Quickly find new period configurations during negotiation.
- On line period variations in control applications.
The **HARTIK** Kernel

### Main Features

- Explicit time management
- EDF-based scheduling
- Bounded priority inversion
- Temporal protection mechanism
- Efficient aperiodic scheduling
- Non-blocking communication
- Flexible interrupt handling
HARTIK Structure

<table>
<thead>
<tr>
<th>Creation and termination</th>
<th>communication and synchronization</th>
<th>utility functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>scheduling</td>
<td>dispatching</td>
<td></td>
</tr>
<tr>
<td>queue management</td>
<td></td>
<td></td>
</tr>
<tr>
<td>context switch</td>
<td>interrupt handling</td>
<td>timer handling</td>
</tr>
</tbody>
</table>

- system calls
- kernel mech.
- ASM

Task types

- **Criticality**
  - HARD (a priory guarantee)
  - SOFT (bandwidth reservation)
  - NRT (non real-time)

- **Activation mode**
  - PERIODIC (time driven)
  - APERIODIC (event driven)
Task scheduling

- HARD tasks
- SOFT tasks
- NRT tasks

CPU

High-Pr. queue

EDF

Low-Pr. queue

priority

Periodic Tasks

$T_i$

RUNNING

READY

IDLE

TIMER

initialization

endcycle

TIMER

TIMER

TIMER
Periodic task code

```c
TASK control()
{
    <local variables>
    <initialization>
    while (!finished) {
        <processing>
        task_endcycle();
    }
}
```

Task states

- SLEEP
  - create
  - activate

- READY
  - signal
  - dispatching
  - preemption
  - wake_up

- WAIT
  - wait

- RUN
  - terminate
  - endcycle

- IDLE
  - sleep

- TIMER
  - wake_up
**Time Management**

- Time is generated by the programmable timer circuit.
- A 16 bit counter is decremented with a frequency of **1191 KHz** (every **0.84 µs**).
- When counter = 0, the timer generates an interrupt and starts counting again.
- Interrupt period (system tick) can range from **0.84 ms** up to \(2^{16} \cdot 0.84 = 55\) ms.

**Timer Initialization**

- To generate interrupts every millisecond
  
  ```c
  #define TFREQ 1191
  ```

- To generate interrupts every arbitrary tick
  
  ```c
  float tick; // resolution in ms
  unsigned int t_const; // timer constant
  
  t_const = tick * TFREQ;
  ```
System Time

A system variable is incremented at every interrupt

```c
unsigned long sys_clock;  // time in ticks
float sys_time;            // time in ms
```

```
sys_time = sys_clock * tick;
```

<table>
<thead>
<tr>
<th>tick</th>
<th>lifetime</th>
</tr>
</thead>
<tbody>
<tr>
<td>50 μs</td>
<td>60 hours</td>
</tr>
<tr>
<td>1 ms</td>
<td>50 days</td>
</tr>
<tr>
<td>10 ms</td>
<td>16 months</td>
</tr>
<tr>
<td>50 ms</td>
<td>7 years</td>
</tr>
</tbody>
</table>

Timer handling routine

- save the context of the running task
- increment system time (sys_clock++)
- if (sys_clock == lifetime) error(LIFETIME_EXPIRED)
- if (sys_clock > deadline) error(DEADLINE_MISS)
- activate idle tasks at the beginning of their period
- if ∃ awaked tasks, invoke the scheduler
- restore the context of the selected task
- return from interrupt (IRET)
Timer Overhead

\[ U_{\text{act}} = U_{\text{lub}} - U_{t} = U_{\text{lub}} - \frac{\sigma}{Q} \]

Actual Utilization Factor

\[ U_{\text{act}} = U_{\text{lub}} - \frac{\sigma}{Q} \]

\( (\sigma = 100 \mu s) \)

- \( Q = 10 \text{ ms} \) \( \Rightarrow \) \( U_{\text{act}} = 0.99 \)
- \( Q = 1 \text{ ms} \) \( \Rightarrow \) \( U_{\text{act}} = 0.9 \)
- \( Q = 200 \mu s \) \( \Rightarrow \) \( U_{\text{act}} = 0.5 \)
HARTIK Applications

- Each HARTIK application starts as a sequential C program.
- The RT environment starts with the `sys_init()` call.
- It ends with the `sys_end()` call.
- Tasks look like functions of `TASK` type.

```c
TASK main()
{
    ...
    sys_init();
    ...
    sys_end();
    ...
}

TASK sensing()
{
    ...
}

TASK control()
{
    ...
}
```
System Initialization

- Define the system tick
- Initialize the real-time environment
- Initialize devices (keyboard, mouse, ...)
- Create tasks
- Activate tasks

Initialization structures

All system parameters are stored in proper structures and have default values:

- **SYS_PARMS**  $sp = \text{BASE\_SYS}$;
- **KEYB_PARMS**  $mp = \text{BASE\_KEYB}$;
- **MOUSE_PARMS**  $mp = \text{BASE\_MOUSE}$;

structure types  structure names  default values
The “MAIN” task

```c
TASK main(void)
{
SYS_PARMS sp = BASE_SYS;
double tick = 1.0;
    sys_def_tick(sp, tick, mSec);
    sys_init(&sp);
    <main body>
    sys_end();
}
```

Task Creation

```
task_create(name, code, type, act, period, m);
```

- **name**: task identification string
- **code**: a pointer to the task code
- **type**: task criticality (**HARD**, **SOFT**, **NRT**)
- **mode**: activation mode (**PERIOD**, **APERIODIC**)
- **period**: task period (in ticks)
- **m**: a pointer to the task model

It returns the process identifier (PID).
Task creation code

```c
MODEL m = BASE_MODEL;
PID p1;
int per = 100;

p1 = task_create("fly", fly, HARD,
                PERIODIC, per, &m);

task_activate(p1);
```

Computation time

- It must be specified in microseconds:

  ```c
  task_def_wcet(m, 50);
  ```

  ⇒ for HARD tasks, it is used to perform a guarantee test at creation time.

  ⇒ for SOFT tasks, it represents the budget allocated to the CBS serving the task:

  $$Q_s = \text{WCET}_i, \quad T_s = T_i$$
Critical exceptions

HARD tasks may cause the following exceptions

- **Guarantee failure** (at creation time)
  
  if \( U_{new} + U > 1 \)

- **Deadline miss** (at runtime)
  
  if \( \text{sys\_clock} > \text{absolute\_deadline} \)

- **WCET violation** (at runtime)
  
  if \( \text{actual\_execution\_time} > \text{WCET} \)

WCET violation

It can be disabled by the instruction:

```
sys_def_nocheck(sp);
```

to be specified before `sys_init(&sp);`

It can be useful during development.
Task argument

- A task may receive an integer argument:
  
  \[
  \text{task\_def\_arg}(m, \ k);
  \]

- It can be used to create multiple instances of the same task with different behavior.

- Each “clone” task evolves independently due to its private state (local variables and stack).

Creating multiple instances

\[
\begin{align*}
\text{task\_def\_arg}(m1, \ arg1); \\
p1 &= \text{task\_create}("fly1", \ fly, \ HARD, \\
&\hspace{1em} \text{PERIODIC, per1, } \&m1); \\
\text{task\_def\_arg}(m2, \ arg2); \\
p2 &= \text{task\_create}("fly2", \ fly, \ HARD, \\
&\hspace{1em} \text{PERIODIC, per2, } \&m2); \\
\text{task\_activate}(p1); \\
\text{task\_activate}(p2);
\end{align*}
\]
Problem with activate

\[ \tau_a \]

activate(\(\tau_1\));
activate(\(\tau_2\));

• Tasks can be activated (or killed) at the same time:

```c
task_def_group(m1, g1);
task_def_group(m2, g1);
task_def_group(m3, g1);
group_activate(g1);
group_kill(g1);
```
Semaphores

HARTIK provides classical semaphores. They can be used for synchronization:

```c
SEM mutex;
mutex = sem_create(0);
```

or mutual exclusion:

```c
SEM sync;
sync = sem_create(1);
```

Critical Sections

Classical blocking behavior:

```c
sem_wait(mutex, BLOCK);
<critical section>
sem_signal(mutex);
```

Used with NON_BLOCK, `sem_wait` does not block and returns 0 on a locked semaphore.
SRP semaphores

Allow to access critical sections according to the Stack Resource Policy:

\[
\text{srp\_wait}(R1);
\]

\[
\text{<critical section>}
\]

\[
\text{srp\_signal}(R1);
\]

We recall that \text{SRP} prevents deadlocks and chained blocking.

---

Message passing paradigm

Every task operates on a private memory space, exchanging messages through channels:

\textit{Channel:} logical link by which two tasks can communicate.

\textit{Message:} set of data having a predefined format.
Communication Ports

- The operating system provides the channel abstraction through the **port** construct.
- A task can use a port to exchange messages by means of two primitives:
  - *send* sends a message to a port
  - *receive* receives a message from a port

Port types

- **STREAM**
  - writer
  - reader

- **MAILBOX**
  - clients
  - server (owner)

- **STICK**
  - producer (owner)
  - consumers
Using a port

Task A

\[ p = \text{port}\_\text{create}(); \]
\[ \text{send}(p, \text{mes}); \]
\[ \text{port}\_\text{delete}(p); \]

Task B

\[ p = \text{port}\_\text{connect}(); \]
\[ \text{receive}(p, \text{mes}); \]
\[ \text{port}\_\text{disconnect}(p); \]

**NOTE:** Task A is the owner and must start first.

Port primitives

- **port\_create**(name, size, num, type, access);
  
  \[ \text{name: identification string} \]
  \[ \text{size: message size (in bytes)} \]
  \[ \text{num: maximum number of messages} \]
  \[ \text{type: STREAM, MAILBOX, STICK} \]
  \[ \text{access: READ, WRITE} \]

  It returns a port identifier.

- **port\_delete**(port\_id);

  Deletes the specified port.
Port primitives

- **port_connect**(name, size, type, access);

  name: stringa di identificazione
  size: message size (in bytes)
  type: STREAM, MAILBOX, STICK
  access: READ, WRITE

  It returns a port identifier.

- **port_disconnect**(port_id);

  Deletes the specified port.

---

Port primitives

- **port_send**(port_id, msg_ptr, sync)

  sends the message pointed by msg_ptr to the port identified by port_id.
  
  sync = BLOCK blocks on a full buffer
  sync = NON_BLOCK returns 0

- **port_receive**(port_id, msg_ptr, sync)

  receives a message from port_id and copies it into the buffer pointed by msg_ptr.
  
  sync = BLOCK blocks on an empty buffer
  sync = NON_BLOCK returns 0
A task creates a port to send messages of 6 bytes. The port can keep up to 8 messages.

```c
TASK writer(void)
{
    PORT p;
    char mes[6];
    p = port_create(“door”, 6, 8, STREAM, WRITE);
    while (condition) {
        build_message(mes);
        port_send(p, mes, BLOCK);
        task_endcycle();
    }
    port_delete(p);
}
```

A task connects to an already opened port to receive messages of 2 bytes.

```c
TASK reader(void)
{
    PORT q;
    char data[2];
    q = port_connect(“door”, 2, STREAM, READ);
    while (condition) {
        if (port_receive(q, data, NON_BLOCK))
            action1();
        else
            action2();
        task_endcycle();
    }
    port_disconnect(q);
}
```
Periodic task communication

- If $T_1 < T_2$, $\tau_1$ puts more messages than $\tau_2$ can read.
- When the buffer is full, $\tau_1$ must proceed with the same rate of $\tau_2$.

Exchanged messages

# of exchanged messages $= \lfloor t \cdot f_i \rfloor$
Buffer Saturation

If $T_1 < T_2$, the buffer saturates when:

$$\left\lfloor \frac{t}{T_1} \right\rfloor - \left\lfloor \frac{t}{T_2} \right\rfloor > N$$

Hence, the tasks proceed at their proper rate while:

$$\left\lfloor \frac{t}{T_1} \right\rfloor - \left\lfloor \frac{t}{T_2} \right\rfloor < \frac{t}{T_1} - \frac{t}{T_2} + 1 < N$$

That is, while:

$$t < (N - 1) \frac{T_1 T_2}{T_2 - T_1}$$

STICK Ports

- If $T_1 < T_2$, $\tau_1$ overwrites previous messages.

Example:

```
T_1
\tau_1
STICK PORT
abcdefg
```

```
T_2 = 2T_1
\tau_2
aceg
```
STICK Ports

- If $T_1 > T_2$, $\tau_2$ reads the same message more than once (messages are not consumed).

**Example:**

\[
\begin{align*}
T_1 &= 2T_2 \\
\tau_1 &\rightarrow \text{STICK PORT} & T_2 \\
\text{abcde} &\rightarrow \text{aabbcddde}
\end{align*}
\]

Blocking on STICK ports

- STICK ports use a semaphore to avoid simultaneous accesses to the internal buffer.
- Long messages may cause long blocking delays on the semaphore.
- A more efficient solution avoids blocking through a buffer replication mechanism.
Cyclic Asynchronous Buffer

- It is a mechanism for exchanging messages among periodic tasks with different rates.
- It avoids memory conflicts by replicating the internal buffers.
- State message semantics: messages are overridden by senders and are not consumed by receivers.

Simultaneous accesses

If a writer task $\tau_W$ arrives while a task $\tau_R$ is reading, the new message is written in a new buffer:
Reading from a CAB

Once written, a message becomes available to the next reader:

\[ \tau_{R1} \quad \text{read } M1 \]
\[ \tau_{R2} \quad \text{read } M1 \quad \text{read } M2 \]
\[ \text{writer} \quad \text{write } M1 \quad \text{write } \quad \text{M2} \]

Accessing a CAB

- CABs are accessed through a memory pointer.
- Hence, a reader is not forced to copy the message in its memory space.
- More tasks can simultaneously read the same message.
- At each instant, a pointer (mrд) points to the most recent message stored in the CAB.
situation at time $t_1$
Dimensioning a CAB

- If a CAB is used by $N$ tasks, to avoid blocking, it must have at least $N + 1$ buffers.
- The $(N+1)$-th buffer is needed for keeping the most recent message in the case all the other buffers are used.

Inconsistency with $N$ buffers

- Assume all buffers are used and $\tau_W$ overwrites the most recent message (M4) with M5.
- If (while $\tau_W$ is writing) $\tau_1$ finishes and requests a new message, it finds the CAB inconsistent.
Writing Protocol

To write a message in a CAB a task must

- ask the CAB for a pointer to a free buffer;
- copy the message into the buffer using the pointer;
- release the pointer to the CAB to make the message accessible to the next reader.

Reading Protocol

To read a message from a CAB a task must

- get the pointer to the most recent message in the CAB;
- process the message through the pointer;
- release the pointer, to allow the CAB to recycle the buffer if it is not used.
CAB Primitives

• **cab_create(cab_name, buf_size, max_buf);**
  
  creates a CAB with max_buf buffers with size buf_size bytes. It returns a global CAB identifier.

• **cab_delete(cab_id);**
  
  deletes the specified CAB.

CAB Primitives

• **cab_reserve(cab_id)**
  
  returns a pointer to write in a free buffer

• **cab_putmes(cab_id, pointer)**
  
  release the pointer after a write operation

• **cab_getmes(cab_id)**
  
  returns a pointer to the most recent message

• **cab_unget(cab_id, pointer)**
  
  release the pointer after a read operation
Writing in a CAB

\[ \cdots \]
\[
p = \text{cab\_reserve}(\text{cab\_id});
\]
\[
\langle \text{copy message in } *p\rangle
\]
\[
\text{cab\_putmes}(\text{cab\_id, } p);
\]
\[ \cdots \]

Reading from a CAB

\[ \cdots \]
\[
p = \text{cab\_getmes}(\text{cab\_id});
\]
\[
\langle \text{process message with } *p\rangle
\]
\[
\text{cab\_unget}(\text{cab\_id, } p);
\]
\[ \cdots \]
Interrupt handling

• The task enables the device to generate interrupts and waits for data:

  <enable device interrupts>
  <wait for interrupt>
  <get data>

• The interrupt activates a driver, which runs at the highest priority, performs data transfer and awakes the user task.
Notes on H-P Interrupts

**Benefit**
- short interrupt latencies

**Problem**
- unbounded and unpredictable delays on user tasks

---

Scheduled Interrupts

- Each interrupt is associated with a handling task (device manager).
- A user task enable interrupts and waits for data.
- At the interrupt arrival, the device manager is activated and scheduled as any other task.
- Finally, the device manager awakes the user task.
Scheduled Interrupts

User task

Kernel Routine

Activate device manager

Device Manager

interrupt

<enable int.>
<wait for int.>
<get data>

interrupt

<signal data>

Scheduled Interrupts

\( \tau_1 \)

\( \tau_2 \)

reader task

Device Manager

Kernel routine

enable interrupt

wait for interrupt

interrupt

signal data ready
Notes on scheduled interrupts

**Benefit**
- small delays on user tasks

**Problem**
- long interrupt latency (possible data loss)

The HARTIK solution

- Each interrupt source is associated with a **fast handler** and a **safe handler**.
- The **fast handler** runs at the highest priority and take the data.
- The **safe handler** is scheduled as any other task, performs more complex operations and awakes the user task.
Interrupt handling in HARTIK

User Task

<enable int.>
<wait for int.>
<get data>

Kernel Routine

call fast handler
activate safe handler

Fast Handler

<signal data>

Interrupt handling in HARTIK

$\tau_1$

$\tau_2$

user task

safe handler

fast handler

enable interrupt

interrupt

signal data ready

wait for interrupt
Notes on scheduled interrupts

Benefits

• high flexibility in device management
• small delays on user tasks
• small interrupt latency

HARTIK is FREE

http://hartik.sssup.it
Developing
Real-Time Control
Applications

Design objectives

• Keep the system complexity low
• Simplify maintainance
• Think of future extensions or modifications
• Guarantee the desired performance in all anticipated workload conditions
General Approach

- Divide a problem into simpler subproblems.
- Organize the software into hierarchical control layers (Top Down vs. Bottom Up).
- Divide each control layer into different interacting modules.

Control Architecture

- Application
- Task level
- Device level
- Kernel level
- HARDWARE
Module Specification

- **Functionality**
  - describe what has to be done

- **Interface**
  - identify inputs and outputs
  - describe the interactions with the other modules

- **Performance**
  - timing constraints to be met

Build a graph with tasks and resources

\[ \begin{align*}
\tau_1 & \rightarrow R1 \rightarrow \tau_2 \\
R2 & \rightarrow \tau_3
\end{align*} \]
Examples of Control Tasks

Force control

\[ F_d - F + K y = \text{force sensor} \]
Force feedback loop

\[
\text{TASK force_control()}
\begin{align*}
&\{ \\
&\quad \text{int } Fd, F; \\
&\quad \text{float } y, K; \\
&\quad \quad \text{while } (1) \{ \\
&\quad \quad \quad \text{K} = \text{read_gain}(); \\
&\quad \quad \quad \text{Fd} = \text{desired force}(); \\
&\quad \quad \quad \text{F} = \text{read_force}(); \\
&\quad \quad \quad \text{y} = \text{K} \times (\text{Fd} - \text{F}); \\
&\quad \quad \quad \text{output}(y); \\
&\quad \quad \quad \text{task_endcycle}(); \\
&\quad \quad \}\}
\end{align*}
\]

Position control

\[
\text{x} \quad \text{0} \quad \text{x}_d \quad \text{x} \\
\text{M} \\
\text{Motor driver} \\
\text{K}_p \quad \text{K}_v \\
\text{x}_d \quad \text{v}_d \quad \text{y} \quad \text{vx} \\
\]
PD regulator

```c
TASK pos_control()
{
int xd, vd, x, v;
float y, Kp, Kv;

while (1) {
    get_gains(&Kp, &Kv);
    get_setpoint(&xd, &vd);
    read_sensors(&x, &v);
    y = Kp*(xd - x) - Kv*(vd - v);
    output(y);
    task_endcycle();
}
}
```

Proximity control

**GOAL:** Control the vehicle to keep the sensor at distance $d > D$ from an object.
Proximity control loop

```c
TASK dist_control()
{
    int d, Ds;
    float y, K;
    while (1) {
        get_parameters(&Ds, &K);
        d = prox_sensor();
        if (d > Ds) y = 0;
        else y = K*(Ds - d);
        output(y);
        task_endcycle();
    }
}
```

Two-level control loop

```
F
---
|     |
|     |
|     |
|     |
K_f

F_d
---
|     |
|     |
|     |
|     |
F

K_v

v_d
---
|     |
|     |
|     |
|     |
K_v

y
---
|     |
|     |
|     |
|     |
Motor
driver

force
sensor
```
Using CABs

Using visual feedback

- An ultrasound sensor is used for speed control
- The camera is used for stear control
Three-level control

visual control 
$T_v = 100$ ms  

distance control 
$T_d = 20$ ms  

motor control 
$T_d = 5$ ms  

$\tau_v$  

$\tau_d$  

$\tau_m$  

camera  

US  

car model $H(z)$  

$\alpha_d$  

$\alpha$  

$\alpha_d$  

$\alpha$  

A simple graphic demo
Object Animation

x \quad \text{current position}
old_x \quad \text{prev. pos.}
\Delta x \quad \text{pos. increment}

\[
x = \text{old}_x = \text{START}_X
\]

compute $\Delta x$

\[
x = x + \Delta x
\]
delete object at old_x
draw object at x

old_x \leftarrow x
task_endcycle()

MAIN start

```c
TASK main(void)
{
  SYS_PARMS parms = BASE_SYS;
  KEYB_PARMS keyb = BASE_KEYB;
  MODEL m = BASE_MODEL;

  sys_def_tick(parms, 1, mSec);
  sys_def_nocheck(parms);
  sys_init(&parms);
  keyb_init(&keyb);
  grx_open(640, 480, 8);
  srand(123);
}
```
MAIN body

do {
    c = keyb_getch(BLOCK);
    if ((c == ' ') && (i < MAX_P)) {
        task_def_arg(m, i);
        task_def_wcet(m, 50);
        pid = task_create("fly", fly, HARD,
                          PERIODIC, per, &m);
        task_activate(pid);
        i++;
    }
} while (c != ESCAPE);

MAIN end

sys_end();
grx_close();
}
References on Real-Time Systems

Most of the topics treated in the course are covered in the book:


Scientific publications on specific topics are listed below.

Task Scheduling

Earliest Due Date


Rate Monotonic


Earliest Deadline First


Schedulability Analysis

Rate Monotonic Analysis


Response Time Analysis


Processor Demand Analysis


Aperiodic Servers

Polling and Deferrable Server


Sporadic Server


Total Bandwidth Server


Constant Bandwidth Server


Resource Access Protocols

Priority Inheritance and Priority Ceiling


Stack Resource Policy


Overload handling

Value-based scheduling


**Robust Scheduling**


**Imprecise computation**


**Job Skipping**


**Elastic scheduling**


**The HARTIK Kernel**


- HARTIK 3.3.1 – Reference Manual. URL: [http://hartik.sssup.it](http://hartik.sssup.it)