Welcome to the Course on
Real-Time Kernels for Microcontrollers:
Theory and Practice

Scuola Superiore Sant’Anna, Pisa
June 23-25, 2008
Course Program

Day 1: Monday, June 23

**Morning:** RT scheduling & resource management (G. Buttazzo)
RT kernels for embedded systems (P. Gai)

**Afternoon:** The FLEX development board (M. Marinoni)
Erika kernel and the OSEK standard (P. Gai)

Day 2: Tuesday, June 24

**Morning:** Developing RT appl. with Erika (P. Gai – M. Marinoni)

**Afternoon:** Laboratory practice (P. Gai – M. Marinoni)

Day 3: Wednesday, June 25

**Morning:** Embedded Systems and Wireless Communication (P. Pagano – G. Franchino)

**Afternoon:** Laboratory practice (P. Pagano – G. Franchino)
Lecture Schedule

09:00 Morning Lecture - Part 1
11:00 Coffee Break
11:15 Morning Lecture - Part 2
13:00 Lunch Break
14:30 Afternoon Lecture - Part 1
16:15 Coffee Break
16:30 Afternoon Lecture - Part 2
18:00 End of Lectures
Real-Time Scheduling and Resource Management

Giorgio Buttazzo

E-mail: buttazzo@sssup.it

Scuola Superiore Sant’Anna
Goal

Provide some background of RT theory for implementing control applications:

- Background and basic concepts
- Modeling real-time activities
- Real-Time Task Scheduling
- Timing analysis
- Handling shared resources
What’s an Embedded System?

⇒ It is a computing system hidden in an object to control its functions, enhance its performance, manage the available resources and simplify the interaction with the user.
What’s special in Embedded Systems?

- **Stringent constraints** on space, weight, energy, cost
  - Scarce resources (processing power, memory)
  - Efficient resource usage at the OS level

- **Interaction with the environment**
  - High responsiveness and timing constraints
  - Schedulability analysis and predictable behavior (RTOS)

- **Robustness** (tolerance to parameter variations)
  - Overload management and system adaptation, to cope with variable resource needs and high load variations.
... and many others

- mobile robot systems
- small embedded devices
  - cell phones
  - videogames
  - smart sensors
  - intelligent toys
Criticality

- QoS management
- High performance
- Safety critical

Timing constraints:
- Soft
- Firm
- Hard
Importance of the RTOS

The Operating System is responsible for:

- managing the **available resources** in an efficient way (memory, devices, energy);
- Enforcing **timing constraints** on computational activities;
- Providing a standard **programming interface** to develop portable applications.
- Providing suitable **monitoring mechanisms** to trace the system evolution to support debugging.
Software Vision

processor

Task

Resource

Environment

D/A

actuators

A/D

sensors
Activation modes

**Periodic tasks:** *(time driven)*

Tasks are automatically activated by the kernel at regular time intervals:

- <read data>
- <process data>
- <write data>
- <wait for next period>

![Diagram of periodic tasks](image)

**Aperiodic tasks:** *(event driven)*

Tasks are activated upon the arrival of an event (interrupt or explicit activation)

- <perform some action>
- <perform some action>

![Diagram of aperiodic tasks](image)
OS support for periodic tasks

```c
while (condition) {
    ___
    ___
    ___
    ___
    wait_for_next_period();
}
```
The IDLE state

- IDLE
- READY
- BLOCKED
- RUNNING

Transitions:
- signal
- activate
- dispatching
- preemption
- wake_up
- end_cycle
- terminate
- Timer
SLEEP state

- SLEEP
- READY
- BLOCKED
- RUNNING
- IDLE

- create
- activate
- signal
- wait
- dispatching
- preemption
- terminate
- end_cycle
- wake_up
- Timer
- end_cycle
Periodic task model

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

\[ r_{i,k} = \Phi_i + (k-1) T_i \]
\[ d_{i,k} = r_{i,k} + D_i \]

- **Period**: \( T_i \)
- **Relative deadline**: \( D_i \)
- **Absolute deadline**: \( D_i = T_i \)

Often:
\[ \Phi_i = 0 \]
Aperiodic task model

- **Aperiodic:** $r_{i,k+1} > r_{i,k}$

- **Sporadic:** $r_{i,k+1} \geq r_{i,k} + T_i$
Periodic Task Scheduling

- We have $n$ periodic tasks: $\{\tau_1, \tau_2, ..., \tau_n\}$

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

**Goal**
- Execute all tasks within their deadlines
- Verify feasibility before runtime

**Assumptions**
- Tasks are execute in a single processor
- Tasks are independent (do not block or self-suspend)
- Tasks are synchronous (all start at the same time)
- Relative deadlines are equal to periods ($D_i = T_i$)
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>

\[ \Delta = \text{GCD} \quad \text{(minor cycle)} \]
\[ T = \text{lcm} \quad \text{(major cycle)} \]

\[
\begin{align*}
\Delta & \geq C_A + C_B \\
\Delta & \geq C_A + C_C
\end{align*}
\]

Guarantee:
Implementation

A

B

A

C

A

B

A

A

A

B

A

C

timer

minor cycle

timer

major cycle

timer

timer
Timeline scheduling

Advantages

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

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 \((B_1, B_2)\) and re-build the schedule:

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

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

<table>
<thead>
<tr>
<th>task</th>
<th>T before</th>
<th>T 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: $\Delta = 25$ \hspace{1cm} $\Delta = 5$

major cycle: $T = 100$ \hspace{1cm} $T = 200$

\[40 \text{ sync. per cycle!}\]
Example
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.
Priority Assignments

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

- **Rate Monotonic (RM):**
  \[ p_i \propto \frac{1}{T_i} \] (static)

- **Earliest Deadline First (EDF):**
  \[ p_i \propto \frac{1}{d_i} \] (dynamic)
  \[ d_{i,k} = r_{i,k} + D_i \]

\[ D_i = T_i \]

\[ r_{i,1} = 0 \]
\[ r_{i,k} \]
\[ r_{i,k+1} \]

\[ t \]
Rate Monotonic (RM)

- Each task is assigned a fixed priority proportional to its rate.
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
A sufficient condition

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

NOTE

If \( U_{lub} < U_p \leq 1 \) we cannot say anything about the feasibility of that task set.
Basic results

Assumptions:

- Independent tasks
- $\Phi_i = 0$
- $D_i = T_i$

In 1973, Liu & Layland proved that a set of $n$ periodic tasks can be feasibly scheduled under RM if

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq n\left(2^{1/n} - 1\right)$$

under EDF if and only if

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$$
RM bound for large $n$

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

for $n \to \infty \quad U_{\text{lub}} \to \ln 2$
Schedulability bound

CPU%
A special case

If tasks have harmonic periods $U_{ub} = 1$.

$$U_p = \frac{2}{4} + \frac{4}{8} = 1$$

\[\tau_1\]
\[\tau_2\]
Schedulability region

The U-space

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

\[ \sum_{i=1}^{n} U_i \leq n(2^{1/n} - 1) \]
Schedulability region

The U-space

\[
\begin{array}{c|cc}
\tau_1 & C_i & T_i \\
\hline
\tau_1 & 3 & 6 \\
\tau_2 & 4 & 9 \\
\end{array}
\]

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

EDF

\[ \tau_1 \]

\[ \tau_2 \]

RM

\[ \tau_1 \]

\[ \tau_2 \]

deadline miss
**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.
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.
For any task $\tau_i$, the longest response time occurs when it arrives together with all higher priority tasks.
The Hyperbolic Bound

• In 2000, Bini et al. proved that a set of $n$ periodic tasks is schedulable with RM if:

$$\prod_{i=1}^{n} (U_i + 1) \leq 2$$
Schedulability region

The U-space

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

\[ \sum_{i=1}^{n} U_i \leq n(2^{1/n} - 1) \]
Schedulability region

The U-space

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

\[ \sum_{i=1}^{n} U_i \leq n(2^{1/n} - 1) \]

\[ \prod_{i=1}^{n} (U_i + 1) \leq 2 \]
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

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.
How to guarantee feasibility?

- Fixed priority: Response Time Analysis (RTA)
- EDF: Processor Demand Criterion (PDC)
Response Time Analysis
[Audsley ‘90]

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

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

- verify if $R_i \leq D_i$
Computing the interference

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

$$ I_{ik} = \left[ \frac{R_i}{T_k} \right] C_k $$

Interference of high priority tasks on $\tau_i$: 

$$ I_i = \sum_{k=1}^{i-1} \left[ \frac{R_i}{T_k} \right] C_k $$
Computing the response time

\[ R_i = C_i + \sum_{k=1}^{i-1} \left( \frac{R_i^{(s-1)}}{T_k} \right) C_k \]

Iterative solution:

\[
\begin{cases}
R_i^0 &= C_i \\
R_i^s &= C_i + \sum_{k=1}^{i-1} \left( \frac{R_i^{(s-1)}}{T_k} \right) C_k \\
& \text{iterate until} \quad R_i^s > R_i^{(s-1)}
\end{cases}
\]
Processor Demand Criterion
[Baruah, Howell, Rosier 1990]

For checking the existence of feasible schedule and for EDF

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

\[ \forall t_1, t_2 > 0, \quad g(t_1, t_2) \leq (t_2 - t_1) \]
The demand in \([t_1, t_2]\) is the computation time of those jobs started at or after \(t_1\) with deadline less than or equal to \(t_2\):

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

For synchronous task sets we can only analyze intervals \([0,L]\)

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

\[ \forall L > 0 \quad \sum_{i=1}^{n} \left( \frac{L - D_i + T_i}{T_i} \right) C_i \leq L \]

**Question**

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

\[ \tau_1 \]

\[ \tau_2 \]

\[ g(0, L) \]

L
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 $L$

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

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

\[ 1 - U \]

for $L > L^*$

\[ g(0,L) \leq G(0,L) < L \]
A set of $n$ periodic tasks with $D \leq T$ is schedulable by EDF if and only if

\[
U < 1 \quad \text{AND} \quad \forall L > 0 \quad \sum_{i=1}^{n} \left( \frac{L - D_i + T_i}{T_i} \right) C_i \leq L
\]

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

\[
H = \text{lcm}(T_1, \ldots, T_n)
\]

\[
L^* = \frac{\sum_{i=1}^{n} (T_i - D_i) U_i}{1 - U}
\]
# Summarizing: RM vs. EDF

<table>
<thead>
<tr>
<th></th>
<th>$D_i = T_i$</th>
<th>$D_i \leq T_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RM</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Suff.</strong>: polynomial $O(n)$</td>
<td><strong>pseudo-polynomial</strong></td>
</tr>
<tr>
<td></td>
<td><strong>LL</strong>: $\sum U_i \leq n(2^{1/n} - 1)$</td>
<td><strong>Response Time Analysis</strong></td>
</tr>
<tr>
<td></td>
<td><strong>HB</strong>: $\prod (U_i + 1) \leq 2$</td>
<td>$\forall i \quad R_i \leq D_i$</td>
</tr>
<tr>
<td></td>
<td><strong>Exact pseudo-polynomial</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>RTA</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$R_i = C_i + \sum_{k=1}^{i-1} \left\lfloor \frac{R_i}{T_k} \right\rfloor C_k$</td>
<td></td>
</tr>
<tr>
<td><strong>EDF</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>polynomial</strong>: $O(n)$</td>
<td><strong>pseudo-polynomial</strong></td>
</tr>
<tr>
<td></td>
<td>$\sum U_i \leq 1$</td>
<td><strong>Processor Demand Analysis</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\forall L &gt; 0, \quad g(0, L) \leq L$</td>
</tr>
</tbody>
</table>
Inter-task communication mechanisms

- Shared memory
- Message passing ports
- Asynchronous buffers
Handling shared resources

Problems caused by mutual exclusion
Critical sections

\[ \tau_1 \]

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

\[ \tau_2 \]

- wait(s)
- \( a = x+1; \)
- \( b = y+2; \)
- \( c = x+y; \)
- signal(s)

Global memory buffer:

- write
- \( \text{int } x; \)
- \( \text{int } y; \)

- read
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 …

$p_1 > p_2$
Priority Inversion

Occurs when a high priority task is blocked by a lower-priority task for an unbounded interval of time.
Resource Access Protocols

Under fixed priorities

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

Under EDF

- 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 CS may also block
Conflict on critical section
Schedule with NPP

\[ P_{CS} = \max \{P_1, \ldots, P_n\} \]
Problem with NPP

$\tau_1$ cannot preempt, although it could

$\tau_1$ blocking, useless
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

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

\( \tau_2 \) 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:

- \( \tau_1 \)
- \( \tau_2 \)
- \( \tau_3 \)

Blocking:

- 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

**Theorem:** $\tau_i$ can be blocked at most once by each lower priority task
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
Example

- $\tau_1$ can be blocked once by $\tau_2$ (on $A_2$ or $C_2$) and once by $\tau_3$ (on $B_3$ or $D_3$)
- $\tau_2$ can be blocked once by $\tau_3$ (on $B_3$ or $D_3$)
- $\tau_3$ cannot be blocked
Schedule with PIP

priority

\( \tau_1 \)
\( \tau_2 \)
\( \tau_3 \)
\( \tau_4 \)

\( P_1 \)
\( P_2 \)
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: $\tau_i$ 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

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

\( t_1: \tau_2 \) is blocked by the PCP, since \( P_2 < C(s_1) \)
PCP properties

Theorem 1
Under PCP, each task can block at most once.

Theorem 2
PCP prevents chained blocking.

Theorem 3
PCP prevents deadlocks.
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
Typical Deadlock

\[ \tau_1 \]

\[ \tau_2 \]

\[ P_1 > P_2 \]

\[ \tau_1 \]

\[ \tau_2 \]

blocked

blocked
Deadlock avoidance with PCP

\( \tau_1 \)

\( \tau_2 \)

\[ P_1 > P_2 \]

\[ C_A = P_1 \]

\[ C_B = P_1 \]

ceiling blocking
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  \((D = T)\)

By LL test:

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

By RTA test: \(\forall i \quad R_i \leq D_i\)

\[
R_i = C_i + B_i + \sum_{k=1}^{i-1} \left( \frac{R_i}{T_k} \right) C_k
\]
Resource Sharing under EDF

The protocols analyzed so far have been originally developed for fixed priority scheduling schemes. However:

- NPP can also be used under EDF
- PIP has been extended under EDF by Spuri (1997).
- PCP has been extended under EDF by Chen-Lin (1990)
- In 1990, Baker proposed a new protocol that works both under fixed and dynamic priorities.
Stack Resource Policy [Baker 1990]

- It works both with fixed and dynamic priority
- It limits blocking to 1 critical section
- It prevents deadlock
- It supports multi-unit resources
- It allows stack sharing
- It is easy to implement
Stack Resource Policy [Baker 90]

- For each resource $R_k$:
  - Maximum units: $N_k$
  - Available units: $n_k$

- For each task $\tau_i$ the system keeps:
  - its resource requirements: $\mu_i(R_k)$
  - a priority $p_i$: $\text{RM } p_i \propto \frac{1}{T_i}$, $\text{EDF } p_i \propto \frac{1}{d_i}$
  - a static preemption level: $\pi_i \propto \frac{1}{D_i}$
Stack Resource Policy [Baker 90]

Resource ceiling

\[ C_k(n_k) = \max_j \left\{ \pi_j : n_k < \mu_j(R_k) \right\} \]

System ceiling

\[ \Pi_s = \max_k \{C_k(n_k)\} \]

SRP Rule

A job cannot preempt until \( p_i \) is the highest and \( \pi_i > \Pi_s \)
Computing Resource Ceilings

\[
\begin{array}{c|c|c|c|c}
\tau_i & D_i & \pi_i & \mu_A & \mu_B \\
\hline
\tau_1 & 10 & 3 & 3 & 0 \\
\tau_2 & 15 & 2 & 1 & 1 \\
\tau_3 & 20 & 1 & 0 & 2 \\
\end{array}
\]
Computing Resource Ceilings

<table>
<thead>
<tr>
<th>( N_R )</th>
<th>( R_A )</th>
<th>3</th>
<th>( R_B )</th>
<th>2</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>( \tau_1 )</th>
<th>( D_i )</th>
<th>( \pi_i )</th>
<th>( \mu_A )</th>
<th>( \mu_B )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \tau_1 )</td>
<td>10</td>
<td>3</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>( \tau_2 )</td>
<td>15</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( \tau_3 )</td>
<td>20</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( C_{R}(3) )</th>
<th>( C_{R}(2) )</th>
<th>( C_{R}(1) )</th>
<th>( C_{R}(0) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( R_A )</td>
<td>0</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>( R_B )</td>
<td>-</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

\[ \text{Di} \]
A task blocks when attempting to preempt
Schedule with PCP

\[ \tau_1 \quad A(3) \quad \tau_2 \quad A(1) \quad B(1) \quad \tau_3 \quad B(2) \]

\[ s_A \quad C(s_A) = P_1 \]

\[ s_B \quad C(s_B) = P_2 \]

A task is blocked when accessing a resource
Lemma

If $\pi_i > C_{R}(n_k)$ then there exist enough units of $R$

1. to satisfy the requirements of $\tau_i$
2. to satisfy the requirements of all tasks that can make preemption on $\tau_i$
Theorem 1

Under SRP, each task can block at most once.

Consider the following scenario where $\tau_1$ blocks twice:

This is not possible, because $\tau_2$ could not preempt $\tau_3$ because, at time $t^*$, $\pi_2 < \Pi_s$
SRP Properties

Theorem 2

If $\pi_i > \Pi_s$ then $\tau_i$ will never block once started.

Proof

Since $\Pi_s = \max \{C_R(n_k)\}$, then there are enough resources to satisfy the requirements of $\tau_i$ and those of all tasks that can preempt $\tau_i$.

Question

If a task can never block once started, can we get rid of the wait / signal primitives?
SRP Properties

Theorem 3

SRP prevents deadlocks.

Proof

From Theorem 2, if a task can never block once started, then no deadlock can occur.
Deadlock avoidance with SRP

\[ \tau_1 \quad \tau_2 \]

\[ \pi_1 > \pi_2 \]
Schedulability Analysis under EDF

When $D_i = T_i$

$$\forall i \quad \sum_{k=1}^{i-1} \frac{C_k}{T_k} + \frac{C_i + B_i}{T_i} \leq 1$$

$B_i$ can be computed as under PCP and refers to the length of longest critical section that can block $\tau_i$. 
EDF Guarantee: PD test \((D_i \leq T_i)\)

Tasks are ordered by decreasing preemption level
Schedulability Analysis under EDF

When $D_i \leq T_i$

A task set is schedulable if $U < 1$ and $\forall L \in D$

$$\forall i \quad B_i + \sum_{k=1}^{n} \left( \frac{L + T_k - D_k}{T_k} \right) C_k \leq L$$

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

$$H = \text{lcm}(T_1, \ldots, T_n) \quad L^* = \frac{\sum_{i=1}^{n} (T_i - D_i)U_i}{1-U}$$
Stack Sharing

Each task normally uses a private stack for:

- saving context (register values)
- managing functions
- storing local variables
Stack Sharing

Why stack cannot be normally shared?

Suppose tasks share a resource: A

Why stack cannot be normally shared?

Suppose tasks share a resource: A
Stack Sharing

Why stack can be shared under SRP?

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

SP2

stack
Saving Stack Size

To really save stack size, we should use a small number of preemption levels.

100 tasks
10 Kb stack per task

\[ \text{stack size} = 1 \text{ Mb} \]

10 preemption levels
10 tasks per group

\[ \text{stack size} = 100 \text{ Kb} \]

\[ \text{stack saving} = 90 \% \]
NOTE on SRP

- SRP for fixed priorities and single-unit resources is equivalent to Higher Locker Priority.

- It is also referred to as Immediate Priority Ceiling.
Non-preemptive scheduling

It is a special case of preemptive scheduling where all tasks share a single resource for their entire duration.

\[ B_i = \max\{C_k : P_k < P_i\} \]
Advantages of NP scheduling

- Reduces runtime overhead
  - Less context switches
  - No semaphores are needed for critical sections

- Reduces stack size, since no more than one task can be in execution.

- Preserves program locality, improving the effectiveness of
  - Cache memory
  - Pipeline mechanisms
  - Prefetch queues
Advantages of NP scheduling

- As a consequence, task execution times are
  - Smaller
  - More predictable
Advantages of NP scheduling

In fixed priority systems can improve schedulability:

\[ U = \frac{2}{5} + \frac{4}{7} \approx 0.97 \]

RM

\( \tau_1 \)

\( \tau_2 \)

NP-RM

\( \tau_1 \)

\( \tau_2 \)

deadline miss
Disadvantages of NP scheduling

- In general, NP scheduling reduces schedulability.
- The utilization bound under non preemptive scheduling drops to zero:

\[ U = \frac{\varepsilon}{T_1} + \frac{C_2}{\infty} \rightarrow 0 \]
Non preemptive scheduling anomalies

double speed

deadline miss
Trade-off solutions

Preemption thresholds

Each task has two priorities:

- Nominal priority (ready priority): used to enqueue the task in the ready queue
- Threshold priority: used for task execution

nominal priority \leq threshold priority
Preemption thresholds

- Nominal pr. = threshold: \( \Rightarrow \) fully preemptive
- Threshold = \( P_{\text{max}} \) \( \Rightarrow \) fully non preemptive

In general:
Trade-off solutions

Tunable Preemptive Systems

- Compute the longest non-preemptive section that allows a feasible schedule.

- Allow preemption only in certain points in the code.