Scheduling

Uwe R. Zimmer - The Australian National University
## References for this chapter

<table>
<thead>
<tr>
<th>Reference</th>
<th>Title</th>
<th>Author(s)</th>
<th>Publisher</th>
<th>Location</th>
<th>Year</th>
</tr>
</thead>
</table>
Scheduling

Motivation

Scheduling in Real-Time Systems

- Concurrency may lead to **non-determinism**.
- **Non-determinism** *may* make it **harder to predict** the timing behaviour.

- **Real-Time-Scheduling schemes reduce non-determinism**.
Scheduling

Deployment

Scheduling in Real-Time Systems

A scheduling scheme provides two features:

- **Ordering** the use of resources (e.g. CPUs, networks) in a live system.
- **Predicting** the worst-case behaviour of the system when the scheduling algorithm is applied.

The prediction can then be used:

- **at compile-run:** to confirm the overall temporal requirements of the application.
- **at run-time:** to permit acceptance of additional usage/reservation requests.
Static versus Dynamic

Scheduling in Real-Time Systems

- **Rigid:**
  - All schedules are set off-line.
  - Full predictability (many high integrity real-time systems).

- **Static:**
  - Schedule relations are statically ordered off-line.
  - Predictable response to disturbances (many real-time systems).

- **Dynamic:**
  - Schedules depend on run-time situation.
  - More flexible, more efficient (most soft real-time systems).
Scheduling phases (non real-time)

- **CPU creation**
- **CPU ready, suspended**
- **CPU blocked**
- **CPU pre-emption or cycle done**

- **CPU admitted dispatch**
- **CPU unblock**
- **CPU suspend (swap-out)**
- **CPU swap-in**
- **CPU swap-out**
- **CPU block or synchronize**

- **CPU termination**
Scheduling phasess (real-time)

- Creation
- Admittance according to schedulability
- Dispatching and Pre-Emption according to deadlines, priorities, or utilities
- Pre-emption or cycle done
- Execution
- Block or synchronize
Assumptions

A simple process model

- The *number* of processes in the system is *fixed*.
- All processes are *periodic* and all periods are *known*.
- All processes are *independent*.
- The task-switching overhead is *negligible*.
- All deadlines are *identical* with the process *cycle times* (periods).
- The worst case execution time is *known* for all processes.
- All processes are *released at once*.

This model can only be applied to a specific group of hard real-time systems. (Extensions to this model will be discussed later in this chapter.)
Real-time scheduling

Task set
Real-time scheduling

Deadlines
Real-time scheduling

Earliest Deadline First (EDF)

1. **Determine** (one of) the process(es) with the earliest deadline.

2. **Execute** this process
   
   2a. *until it finishes.*
   
   2b. *until another process’ deadline is found earlier* then the current one.

- Pre-emptive scheme.
- Dynamic scheme,
  
  since the dispatched process is selected at run-time, due to the current deadlines.
Real-time scheduling: Earliest Deadline First

Execute EDF schedule \(\Rightarrow\) Works!

If multiple deadlines coincide, other means are needed to select a process, i.e.

- Avoid unnecessary task switches.
- Dispatch by task id (out of the currently qualifying processes).
Real-time scheduling: Earliest Deadline First

Times to deadlines

Gives an idea how “critical” the schedule is.
Real-time scheduling: Earliest Deadline First

Worst case response times

Response time: Time from schedule request to process completion.

In the example: Worst case response times are identical to cycle times.
Real-time scheduling: Earliest Deadline First

Maximal utilization

If deadlines $D_i$ are identical to cycle times $T_i$ for each task $i$ then:

The maximal utilization for EDF becomes: $\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$ (sufficient and necessary test)
Real-time scheduling

**Fixed Priority Scheduling (FPS), rate monotonic**

1. Each process is assigned a fixed priority according to its cycle time:
   \[ T_i < T_j \Rightarrow P_i > P_j \]

2. At run-time: dispatch the runnable process with the highest priority.

- Pre-emptive scheme
- Static scheme, since the order dispatch order of processes is fixed and calculated off-line.

- Rate monotonic ordering is **optimal**
  (in the framework of fixed priority schedulers):
  if a process set is schedulable under an FPS-scheme, then it is also schedulable under FPS with rate monotonic priorities.
Real-time scheduling: Fixed Priority Scheduling

Task set
Real-time scheduling: Fixed Priority Scheduling

Execute FPS schedule \(\text{Fails!}\)
Real-time scheduling: Fixed Priority Scheduling

Execute FPS schedule \(\rightarrow\) Fails!
Real-time scheduling: Fixed Priority Scheduling

Maximal utilization

\[ U \equiv \frac{\sum_{i=1}^{n} \frac{C_i}{T_i}}{N} \leq N \left(2^{\frac{1}{N}} - 1\right) \equiv U_{\max} \]

with \( C_i \) the computation time and \( T_i \) the length of the period for task \( i \) out of \( N \) tasks and assuming that the deadline \( D_i = T_i \)

Sufficient, yet not necessary test
Real-time scheduling: Fixed Priority Scheduling

Execute FPS schedule → Fails!

Utilization test fails, schedulability not guaranteed.

\[ U \equiv \sum_{i=1}^{n} \frac{C_i}{T_i} = 1 > 0.779 \approx N\left(2^{\frac{1}{N}} - 1\right) \equiv U_{\text{max}} \]
Real-time scheduling: Fixed Priority Scheduling

Reduced task set

\[ U \equiv \sum_{i=1}^{n} \frac{C_i}{T_i} = \frac{6}{16} + \frac{3}{12} + \frac{1}{4} = 0.875 > 0.779 \approx N \left(2^{\frac{1}{N}} - 1\right) \equiv U_{\text{max}} \]

Utilization test fails, schedulability not guaranteed.
Execute FPS schedule \( \Rightarrow \) Works!

\[ U \equiv \sum_{i=1}^{n} \frac{C_i}{T_i} = \frac{6}{16} + \frac{3}{12} + \frac{1}{4} = 0.875 > 0.779 \approx N(2^{\frac{1}{N}} - 1) \equiv U_{\text{max}} \]

\( U \) Utilization test fails, schedulability not guaranteed.
Real-time scheduling: Fixed Priority Scheduling

Execute FPS schedule \(\Rightarrow\) Works!

\[ U \equiv \sum_{i=1}^{n} \frac{C_i}{T_i} = \frac{6}{16} + \frac{3}{12} + \frac{1}{4} = 0.875 > 0.779 \approx N(2^{\frac{1}{N}} - 1) \equiv U_{\text{max}} \]

\(\bowtie\) Utilization test fails, schedulability not guaranteed.
Real-time scheduling: Fixed Priority Scheduling

Further reduced task set

\[ U \equiv \sum_{i=1}^{n} \frac{C_i}{T_i} = \frac{4}{16} + \frac{3}{12} + \frac{1}{4} = 0.75 < 0.779 \approx N \left( \frac{1}{N} - 1 \right) \equiv U_{\text{max}} \]

Utilization test succeeds, schedulability guaranteed.
Real-time scheduling: Fixed Priority Scheduling

Execute FPS schedule \(\implies\) Works!

\[
U = \sum_{i=1}^{n} \frac{C_i}{T_i} = \frac{4}{16} + \frac{3}{12} + \frac{1}{4} = 0.75 < 0.779 \approx N\left(2^{\frac{1}{N}} - 1\right) \equiv U_{\text{max}}
\]

\(\therefore\) Utilization test succeeds, schedulability guaranteed.
Real-time scheduling: Fixed Priority Scheduling

Execute FPS schedule \( \Rightarrow \) Works!

\[
U \equiv \sum_{i=1}^{n} \frac{C_i}{T_i} = \frac{4}{16} + \frac{3}{12} + \frac{1}{4} = 0.75 < 0.779 \approx N\left(\frac{1}{N} - 1\right) \equiv U_{\text{max}}
\]

Utilization test succeeds, schedulability guaranteed.
Real-time scheduling: Fixed Priority Scheduling

Worst case response times

for the highest priority task: $R_3 = C_3$
Real-time scheduling: Fixed Priority Scheduling

Worst case response times

for others tasks: $R_j = C_j + l_j$ (interference from higher priority tasks)
Real-time scheduling: Fixed Priority Scheduling

Worst case response times

for others tasks: 
\[
R_j = C_j + \sum_{k > j} \left[ \frac{R_j}{T_k} \right] \cdot C_k
\]
Real-time scheduling: Fixed Priority Scheduling

Response time analysis

\[ R_j = C_j + \sum_{k > j} \left( \frac{R_j}{T_k} \right) \cdot C_k \]

Fixed-point equation

Recurrent form: \( R_j^{t+1} = C_j + \sum_{k > j} \left( \frac{R_j^t}{T_k} \right) \cdot C_k \) with: \( R_j^0 = C_j \)

Iterate the recurrent form until: \( R_j^{t+1} = R_j^t \) or \( R_j^{t+1} > D_j \)
Real-time scheduling: Earliest Deadline First

Response time analysis

The worst case for Earliest Deadline First is *not necessarily* when all tasks are released at once!

- All possible release combinations in a full hyper-cycle need to be considered!

- The response times are bounded by the cycle times as long as the maximal utilization is $\leq 1$.
- Other tasks need to be considered only, if their deadline is closer or equal to the current task.
Real-time scheduling: Earliest Deadline First

Response time analysis

\[
R_j(a) = \left\lfloor \frac{a}{T_j} + 1 \right\rfloor C_j + \sum_{k \neq j, \min} \left\{ \left\lfloor \frac{R_j(a)}{T_k} \right\rfloor, \left\lfloor \frac{a + T_j - T_k}{T_j} \right\rfloor + 1 \right\} \cdot C_k
\]
Real-time scheduling: Earliest Deadline First

Response time analysis

\[ R_j(a) = \left\lfloor \frac{a}{T_j} + 1 \right\rfloor C_j + \sum_{k \neq j} \min_{\text{max}} \left\{ \left\lfloor \frac{R_j(a)}{T_k} \right\rfloor, \max \left\{ 0, \left\lfloor \frac{a + T_j - T_k}{T_j} \right\rfloor + 1 \right\} \right\} \cdot C_k \]

Fixed-point equation

Recurrent form:

\[ R_j^{t+1}(a) = \left\lfloor \frac{a}{T_j} + 1 \right\rfloor C_j + \sum_{k \neq j} \min_{\text{max}} \left\{ \left\lfloor \frac{R_j^t(a)}{T_k} \right\rfloor, \max \left\{ 0, \left\lfloor \frac{a + T_j - T_k}{T_j} \right\rfloor + 1 \right\} \right\} \cdot C_k \]

with: \( R_j^0 = a + C_j \)

Iterate until: \( R_j^{t+1}(a) = R_j^t(a) \) or \( R_j^{t+1}(a) > a + D_j \)
Real-time scheduling: Earliest Deadline First

Response time analysis

\[
R_j(a) = \left(\frac{a}{T_j} + 1\right)C_j + \sum_{k \neq j} \min \left\{ \left[ \frac{R_j(a)}{T_k} \right], \max \left\{ 0, \left[ \frac{a + T_j - T_k}{T_j} \right] + 1 \right\} \right\} \cdot C_k
\]

\( \text{Fixed point equation} \)

\[
R_j = \max \{ R_j(a) - a \} \quad \text{where} \quad A = \text{scm}(T_i)
\]
Real-time scheduling: Fixed Priority Scheduling

Worst case response times

\[ R_3 = 1 \leq 4 \checkmark; \quad R_2 = 4 \leq 12 \checkmark; \quad R_1 = 10 \leq 16 \checkmark; \quad \sum_{i=1}^{n} \frac{C_i}{T_i} \leq N \left( 2^\frac{1}{N} - 1 \right) \checkmark \]
Real-time scheduling: Fixed Priority Scheduling

Worst case response times

\[ R_3 = 1 \leq 4 \checkmark; \quad R_2 = 4 \leq 12 \checkmark; \quad R_1 = 12 \leq 16 \checkmark; \quad \sum_{i=1}^{n} \frac{C_i}{T_i} > N(2^{\frac{1}{N}} - 1) \]
Real-time scheduling: Fixed Priority Scheduling

Worst case response times

\[ R_3 = 1 \leq 4 \checkmark; \quad R_2 = 4 \leq 12 \checkmark; \quad R_1 = 19 > 16 \times; \quad \sum_{i=1}^{n} \frac{C_i}{T_i} > N(2^{\frac{1}{N}} - 1) \times \]
Real-time scheduling: Earliest Deadline First

Worst case response times

Testing all combinations in a hyper-cycle:

\[ R_j = \max \{ R_j(a) - a \} \quad \text{where} \quad A = \text{scm}(T_i) \]
Real-time scheduling: Earliest Deadline First

Worst case response times

\[ R_3 = 4 \leq 4 \checkmark; \quad R_2 = 12 \leq 12 \checkmark; \quad R_1 = 16 \leq 16 \checkmark; \quad \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1 \checkmark \]
Real-time scheduling: Earliest Deadline First

Worst case response times

\[
R_3 = 1 \leq 4 \checkmark; \quad R_2 = 8 \leq 12 \checkmark; \quad R_1 = 12 \leq 16 \checkmark; \quad \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1 \checkmark
\]
Real-time scheduling: Earliest Deadline First

Worst case response times

\[
R_3 = 1 \leq 4 \checkmark; \quad R_2 = 6 \leq 12 \checkmark; \quad R_1 = 10 \leq 16 \checkmark; \quad \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1 \checkmark
\]
Real-time scheduling: Comparison

Response Time Analysis

<table>
<thead>
<tr>
<th>Fixed Priority Scheduling</th>
<th>Earliest Deadline First</th>
</tr>
</thead>
<tbody>
<tr>
<td>Utilization Test</td>
<td>Response Times ( {R_i} )</td>
</tr>
<tr>
<td>[ \sum_{i=1}^{n} \frac{C_i}{T_i} \leq N(2^{\frac{1}{N}} - 1) ]</td>
<td>[ C_j + \sum_{k&gt;j} \left\lfloor \frac{R_j}{T_k} \right\rfloor \cdot C_k ]</td>
</tr>
<tr>
<td>((T_i, C_i)) = {(16, 8), (12, 3), (4, 1)}</td>
<td>(\times)</td>
</tr>
<tr>
<td>((T_i, C_i)) = {(16, 6), (12, 3), (4, 1)}</td>
<td>(\times)</td>
</tr>
<tr>
<td>((T_i, C_i)) = {(16, 4), (12, 3), (4, 1)}</td>
<td>(\checkmark)</td>
</tr>
</tbody>
</table>
Real-time scheduling: Comparison

**Fixed Priority Scheduling ↔ Earliest Deadline First**

- EDF can handle higher (full) utilization than FPS.
- FPS is easier to implement and implies less run-time overhead

- Graceful degradation (resource is over-booked):
  - FPS: processes with lower priorities will always miss their deadlines first.
  - EDF: any process can miss its deadline and can trigger a cascade of failed deadlines.

- Response time analysis and utilization tests:
  - FPS: $O(n)$ utilization test — response time analysis: fixed point equation
  - EDS: $O(n)$ utilization test — response time analysis: fixed point equation in hyper-cycle
Real-World Extension

Simplistic Assumptions

- Tasks are periodic
- Deadlines are identical with task’s period time ($D = T$)
- Tasks are independent
- Pre-emptive scheduling
- Worst case execution times are known
Real-World Extension

More Realistic Assumptions

- Tasks are *periodic*
  - we will introduce *sporadic* and *aperiodic* processes
- *Deadlines* are identical with *task’s period time* \(D = T\)
  - we will introduce *arbitrary deadlines*
- Tasks are *independent*
  - we will introduce schedules for *interacting tasks*
- Pre-emptive scheduling
  - we will introduce (briefly) *cooperative scheduling*
- Worst case *execution times* are *known*
  - we will introduce *fault tolerant scheduling*
Real-World Extension

More Realistic Assumptions

- Tasks are periodic
  - we will introduce sporadic and aperiodic processes

- Deadlines are identical with task’s period time \((D = T)\)
  - we will introduce arbitrary deadlines

- Tasks are independent
  - we will introduce schedules for interacting tasks

- Pre-emptive scheduling
  - we will introduce (briefly) cooperative scheduling

- Worst case execution times are known
  - we will introduce fault tolerant scheduling
Scheduling

Sporadic and Aperiodic Processes

Hard real-time tasks
Sporadic and Aperiodic Processes

FPS for hard real-time tasks

(4, 1)

(16, 7)

(T_p, C_i)

© 2015 Uwe R. Zimmer, The Australian National University
Scheduling

Sporadic and Aperiodic Processes

Introducing a soft real-time task

Sporadic / aperiodic task set to lowest priority.
Scheduling

Sporadic and Aperiodic Processes

**FPS lowest priority for soft real-time task**

Sporadic / aperiodic task does not interfere with hard real-time tasks.

- Response times for sporadic / aperiodic task can be large.
Setting a **deferrable server** task as a proxy for sporadic / aperiodic tasks on highest priority level.
Scheduling

Sporadic and Aperiodic Processes

**FPS server task as normal task**

Hard real-time tasks are still schedulable with the server task deploying its full length.
Scheduling

Sporadic and Aperiodic Processes

Sporadic task utilizing deferrable server

Schedule must also work will less interference.

Deferrable server task only deploying if there are requests from the sporadic / aperiodic task.
Sporadic and Aperiodic Processes

Sporadic task utilizing sporadic server

A sporadic server only replenishes after a fixed time after its actual deployment.

- Interference level less or equal to a deferrable server.
- Minimal inter-arrival-times knowledge is employed.
Sporadic and Aperiodic Processes

FPS with dual priorities

Start the sporadic / aperiodic tasks on high priority and demote them in time for the hard real-time tasks to complete (dynamic scheduling scheme).

Pushes the hard real-time tasks to their deadlines.
Introducing an EDF server

The EDF equivalent to a deferrable server: a periodic server task with an immediate deadline.
Sporadic and Aperiodic Processes

Sporadic task utilizing EDF server

Swift response times for the sporadic / aperiodic tasks with deadlines pushed to their limits.
Scheduling

Sporadic and Aperiodic Processes

Earliest Deadline Last (EDL) for sporadic tasks

Earliest Deadline Last scheduling (while still keeping all deadlines) when sporadic / aperiodic tasks are to be scheduled.

Deadlines explicitly pushed to their limits during the EDL phases.
Real-World Extension

More Realistic Assumptions

- Tasks are periodic
  - we will introduce sporadic and aperiodic processes
- Deadlines are identical with task’s period time \((D = T)\)
  - we will introduce arbitrary deadlines
- Tasks are independent
  - we will introduce schedules for interacting tasks
- Pre-emptive scheduling
  - we will introduce (briefly) cooperative scheduling
- Worst case execution times are known
  - we will introduce fault tolerant scheduling
Tasks with arbitrary deadlines

Tasks with $D < T$
(Deadline earlier than cycle time)

In case of fixed priority scheduling (FPS):

Change from:

Rate Monotonic Priority Ordering (RMPO)

to:

Deadline Monotonic Priority Ordering (DMPO)

Lemma:

Any task set $Q$ which is schedulable by a FPS scheme $W$, is also schedulable under DMPO.
Proof of DMPO optimality

Swap two priorities out of $W$ which violate DMPO:

1. $t_i$, $t_j$ are two tasks in $Q$ with $P_i > P_j$ and $D_i > D_j$ in $W \nRightarrow \neg \text{DMPO}$

2. Generate $W'$ by swapping $P_i$ and $P_j$ $\overset{\text{PP}}{\Rightarrow} (P_i' < P_j') \land (D_i > D_j) \overset{\text{DMPO}}{\Rightarrow} W'$

3. $W'$ schedules $Q$ because:

   3a. All $t_k \in Q$ with $P_k > P_i$ or $P_k < P_j$ are unaffected.

   3b. $t_j$ is schedulable in $W'$ because $P_j' > P_j \Rightarrow R_j' \leq R_j \leq D_j \Rightarrow R_j' \leq D_j$

   3c. $t_i$ is schedulable in $W'$ because:

       in $W$: $R_j \leq D_j < D_i \leq T_i \Rightarrow R_j < T_i$ meaning that $t_i$ interfered only once with $t_j$

       also: $t_j$ released once in $R_j$, and $R_i < R_j$

       $\overset{\text{RR}}{\Rightarrow} W': t_j$ interferes only once with $t_i \overset{R_j'}{\Rightarrow} R_j' \leq D_j < D_i \Rightarrow R_j' < D_i$
Proof of DMPO optimality

Swap two priorities out of $W$ which violate DMPO:

$$W$$

in $W$: $D_j < D_i \leq T_i \Rightarrow D_j < T_i$
Proof of DMPO optimality

Swap two priorities out of W which violate DMPO:

in W: \( R_j \leq D_j < D_i \leq T_i \Rightarrow R_j < T_i \) s.t. \( t_i \) interfered only once with \( t_j \)
also: \( t_j \) released once in \( R_j \), and \( R_i < R_j \)
Proof of DMPO optimality

Swap two priorities out of $W$ which violate DMPO:

in $W$: $R_j \leq D_j < D_i \leq T_i \Rightarrow R_j < T_i \iff t_i$ interfered only once with $t_j$
also: $t_j$ released once in $R_j$, and $R_i < R_j$

in $W'$:
Proof of DMPO optimality

Swap two priorities out of $W$ which violate DMPO:

In $W$: $R_j \leq D_j < D_i \leq T_i \Rightarrow R_j < T_i \Leftrightarrow t_i$ interfered only once with $t_j$

also: $t_j$ released once in $R_j$, and $R_i < R_j$

In $W'$: $t_j$ interferes only once with $t_i \Leftrightarrow R_i' = R_j \leq D_j < D_i \Rightarrow R_i' < D_i \checkmark$
Proof of DMPO optimality

Swap all priorities out of W which violate DMPO:

Swap all $t_i, t_j$ in Q, with $(P_i > P_j) \land (D_i > D_j)$ in W resulting in all $t_i, t_j$ in Q with $P_i > P_j$ to have $D_i < D_j$

Constituting the DMPO scheme

Since each swapping operation keep schedulability, the resulting DMPO scheme is also schedulable. ✓

Deadline monotonic ordering is optimal:

(if a process set is schedulable under an FPS-scheme, then it is also schedulable under FPS with deadline monotonic priorities.)
Tasks with arbitrary deadlines

Tasks with $D > T$
(Deadline later than cycle time)

Assumption: every task $t_i$ is released only after the former release of $t_i$ is completed.

❖ In case that $R_i > T_i$ for a specific scheduling situation, the following release of task $t_i$ is delayed by $R_i - T_i$.
❖ Mind that $R_i > T_i$ cannot hold for all release situations, otherwise the task is not schedulable.
❖ The worst case response time $[R_i]$ might thus be longer than $T_i$ but must still be shorter than $D_i$. 
Tasks with arbitrary deadlines

Tasks with $D > T$
(Deadline later than cycle time)

Assumption: every task $t_i$ is released only after the former release of $t_i$ is completed.

Since the response time $R$ can now be potentially greater than the cycle time $T$:

More than one release $q$ of the task $t_i$ needs to be considered:

$$R_i(q) = B_i + qC_i + \sum_{k > i} \left[ \frac{R_i(q)}{T_k} \right] C_k \text{ where } \forall q | R_i(q) - (q - 1) T_i \leq D_i$$

$B_i$ is the blocking time; $q$ is the number of releases.

$$R_i = \max \left\{ R_i(q) - (q - 1) T_i \mid q \in \{1...q_{\text{max}}\} \right\} \text{ and } q_{\text{max}} = \left\{ q \mid \frac{R_i(q)}{q} \leq T_i \right\}$$
Real-World Extension

More Realistic Assumptions

- Tasks are periodic

  we will introduce sporadic and aperiodic processes

- Deadlines are identical with task’s period time \( D = T \)

  we will introduce arbitrary deadlines

- Tasks are independent

  we will introduce schedules for interacting tasks

- Pre-emptive scheduling

  we will introduce (briefly) cooperative scheduling

- Worst case execution times are known

  we will introduce fault tolerant scheduling
Interdependent tasks

Independent tasks

Deadlines identical to cycle times \( \Rightarrow \) DMPO or RMPO.
Scheduling

Interdependent tasks

Independent tasks

Schedulable under DMPO or RMPO.
Interdependent tasks

Task dependencies

Lock requests by two tasks.
Interdependent tasks

Task dependencies

The lower priority task blocks the higher priority task. (note that the blue task is unaffected.)

_priority inversion
Interdependent tasks

Priority inheritance

Task $t_i$ inherits priority $P_k$ of task $t_k$ if:

1. $P_i < P_k$.
2. Task $t_i$ has locked a resource $Q$.
3. Task $t_k$ is blocked waiting for the release of resource $Q$. 
Interdependent tasks

Priority inheritance

Maximal blocking time for task $t_i$: $B_i = \sum_{r=1}^{R} \text{usage}(r, i) C(r)$ with:

- $R$ denoting the number of critical sections.
- $\text{usage}(r, i)$ being a boolean function returning “1” for true and indicating the $r$ is used by:
  - at least one $t_j$ with $P_j < P_i$
  - and at least one $t_k$ with $P_k \geq P_i$
- $C(r)$ denoting the worst case computation time in critical section $r$

Each task can only be blocked once for each employed resource!
Interdependent tasks

Priority inheritance

The lower priority task is promoted to the priority of the blocked task.

The task on priority 2 is blocked due to priority inheritance.
Interdependent tasks

Without priority inheritance

The lower priority task blocks the higher priority task. (note that the blue task is unaffected.)

Priority inversion
Interdependent tasks

Task dependencies with multiple locks
Interdependent tasks

Task dependencies with multiple locks

Scheduling by DMPO/RMPO results in blocking for the higher priority tasks.
Interdependent tasks

Task dependencies with multiple locks

Scheduling by DMPO/RMPO with priority inheritance does not improve the result.
Interdependent tasks

Circular task dependencies
Scheduling by DMPO/RMPO results in deadlock.
(Priority inheritance does not make a difference for blocked tasks.)
Interdependent tasks

Immediate ceiling priority protocol (POSIX, Ada, RT-Java)

- Each task $t_i$ has a static priority $P_i$.
- Each resource $R_k$ has a static ceiling priority $C_k$:

$$C_k = \max \{ \text{employ}(i,k) \cdot P_i \}$$

with employ $(i,k)$ being a boolean function returning “1” for true if task $t_i$ employs resource $R_k$.
- Each task $t_i$ has a dynamic priority $P^D_i$:

$$P^D_i = \max \{ P_i, \max \{ \text{locked}(i,k) \cdot C_k \} \}$$

with locked $(i,k)$ being a boolean function returning “1” for true if task $t_i$ holds resource $R_k$. 
Interdependent tasks

Ceiling Priority Protocol

Avoids the deadlock!
Interdependent tasks

Immediate ceiling priority protocol (POSIX, Ada, RT-Java)

Implications:

- Tasks are dispatched only if all employed resources are available.
- Deadlocks are prevented (no hold and wait).
- Number of context switches are reduced.
Interdependent tasks

**Immediate ceiling priority protocol** (POSIX, Ada, RT-Java)

Maximal blocking time: \( B_i = \max \{ \text{usage}(r, i) \cdot C(r) \}_r \)

- \( R \) denoting the number of critical sections.
- \( \text{usage}(r, i) \) being a boolean function returning “1” for true and indicating that \( r \) is used by:
  - at least one \( t_j \) with \( P_j < P_i \).
- \( C(r) \) denoting the worst case computation time in critical section \( r \)

Each task can only be blocked once by one lower priority task!
Real-World Extension

More Realistic Assumptions

- Tasks are periodic
  - we will introduce sporadic and aperiodic processes
- Deadlines are identical with task’s period time \( D = T \)
  - we will introduce arbitrary deadlines
- Tasks are independent
  - we will introduce schedules for interacting tasks
- Pre-emptive scheduling
  - we will introduce (briefly) cooperative scheduling
- Worst case execution times are known
  - we will introduce fault tolerant scheduling
Non pre-emptive scheduling

In pre-emptive scheduling:

- Maximal individual blocking times $B_i$ can be determined for each task $t_i$ by employing a priority ceiling protocol.
- Maximum overall blocking time $B_{\text{max}} = \max\{B_i\}$.

Cooperative Scheduling

- Every task $t_i$ is divided in $k$ non pre-emptive blocks of $C_{i_k} \leq B_{\text{max}}$.
- All critical sections are completely enclosed in a single block $C_{i_k}$.
- Every task calls a “de-scheduling” routine at the end of each block, i.e. “offering” a task switch.
Non pre-emptive scheduling

Cooperative Scheduling

Implications:

- Number of task switches is reduced.
- Caches, pre-fetching, and pipelines are more efficient.
- Execution times are (a bit) easier to predict.
- Schedules are simpler.
- Interdependent task sets are schedulable deadlock free by design.
Non pre-emptive scheduling

Cooperative Scheduling

Response times:

\[ R_i = R_i^n + F_i \quad \text{with} \quad R_i^{k+1} = B_{max} + C_i - F_i + \sum_{j > i} \left[ \frac{R_j^k}{T_j} \right] C_j \]

with \( F_i \) the execution time of the final block.

For the simplified case of \( C = C_i = C_j = F_i = B_{max} \):

\[ R_i = R_i^n \quad \text{with} \quad R_i^{k+1} = C + \sum_{j > i} \left[ \frac{R_j^k}{T_j} \right] C \]

For the further simplified case of \( \forall i: T = T_i \):

\[ R_i = C + \sum_{j > i} C \]
Non pre-emptive scheduling

Cooperative Scheduling

Considerations:

- Code block division need to be done thoroughly.
- Additional protection against misbehaving (non-cooperative) tasks:
  - Scheduler **pre-empts** tasks (deferred pre-emption), which fail to offer a ‘de-schedule’ themselves.
- Due to a central $B_{max}$, additional tasks need to be engineered to participate in a specific cooperative schedule.
- Requires that a value $B_{max}$ can be accepted by all tasks.
  - Short and reactive tasks are excluded or treated separately.
Real-World Extension

More Realistic Assumptions

- Tasks are periodic
  - we will introduce sporadic and aperiodic processes
- Deadlines are identical with task’s period time \( D = T \)
  - we will introduce arbitrary deadlines
- Tasks are independent
  - we will introduce schedules for interacting tasks
- Pre-emptive scheduling
  - we will introduce (briefly) cooperative scheduling
- Worst case execution times are known
  - we will introduce fault tolerant scheduling
Fault Tolerance

Exceptions and Recoveries

Task $t_i$ needs extra CPU-time $C_i^f$ for error recovery or exception handling and the minimum inter-arrival time between faults is $T_f$:

$$R_i = B_i + C_i + \sum_{j > i} \left[ \frac{R_i}{T_j} \right] C_j + \max_k \left\{ \left[ \frac{R_i}{T_f} \right] C_k^f \right\}_{k \geq i}$$

If error recovery is performed at the highest priority:

$$R_i = B_i + C_i + \sum_{j > i} \left[ \frac{R_i}{T_j} \right] C_j + \max_k \left\{ \left[ \frac{R_i}{T_f} \right] C_k^f \right\}_{k}$$
Scheduling

Real-World Extension

More Realistic Assumptions

• Tasks are periodic
  - we will introduce sporadic and aperiodic processes

• Deadlines are identical with task’s period time \( (D = T) \)
  - we will introduce arbitrary deadlines

• Tasks are independent
  - we will introduce schedules for interacting tasks

• Pre-emptive scheduling
  - we will introduce (briefly) cooperative scheduling

• Worst case execution times are known
  - we will introduce fault tolerant scheduling
General scheduling methods

Some task sets can be scheduled by introducing offsets to the release times, yet …

Without any further restrictions this problem is \( \text{NP-hard} \)

By introducing further assumptions about cycle time granularity and associated deadlines:

Schedulability analysis’ complexity can be reduced to polynomial.

e.g. Restrict cycle times to powers of two of a base time.
Language support

Ada provides:

- Task and interrupt priorities (static, dynamic, active).
- Task attributes.
- Prioritized entry queues.
- Priority ceiling locking (ICPP).
- Schedulers
  (FPS with FIFO within priorities (pre-emptive), Round Robin, EDF).
- Task execution time measurements.

Ada does currently not provide:

- Sporadic servers
package System is

  subtype Any_Priority is Integer range implementation-defined;

  subtype Priority is Any_Priority range Any_Priority'First .. implementation-defined;

  subtype Interrupt_Priority is Any_Priority range Priority'Last + 1 .. Any_Priority'Last;

  Default_Priority : constant Priority := (Priority'First + Priority'Last) / 2;

end System;

package Ada.Dynamic_Priorities is

  procedure Set_Priority (Priority : in System.Any_Priority;
                          T : in Ada.Task_Identification.Task_ID
                          := Ada.Task_Identification.Current_Task);

  function Get_Priority (T : Ada.Task_Identification.Task_ID
                         return System.Any_Priority;

end Ada.Dynamic_Priorities;
Language support

POSIX provides:

- Threads and interrupt priorities (static, dynamic, active).
  - Threads can be ‘system contented’ or
  - ‘process contented’ (priority scheduling unclear in this case).
- Prioritized message queues.
- Priority ceiling locking (ICPP).
- Schedulers, priority based with at least:
  - FIFO, Round-Robin, Sporadic Server, possibly others.
- Timers.
Summary

Scheduling

• **Basic real-time scheduling**
  - Fixed Priority Scheduling (FPS) with Rate Monotonic (RMPO) and Deadline Monotonic Priority Ordering (DMPO).
  - Earliest Deadline First (EDF).

• **Real-world extensions**
  - Aperiodic, sporadic, soft real-time tasks.
  - Deadlines different from period.
  - Synchronized talks (priority inheritance, priority ceiling protocols).
  - Cooperative and deferred pre-emption scheduling.
  - Fault tolerance in terms of exception handling considerations.

• **Language support**
  - Ada, POSIX