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**.

**Motivation**

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

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).

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).

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

(16, 0)
(12, 3)
(4, 1)

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 than the current one.

Pre-emptive scheme.
Dynamic scheme, since the dispatched process is selected at run-time, due to the current deadlines.

Deadlines

(16, 0)
(12, 3)
(4, 1)

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

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)

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

Execute FPS schedule fails!

(Maximal utilization)

\[ U = \sum_{i=1}^{N} \frac{C_i}{T_i} \leq N \left( \frac{1}{N} - 1 \right) = U_{\text{MAX}} \]

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

Maximally utilizable number of processes \( N \): sufficient, yet not necessary test.

Maximal utility

Number of processes \( N \)
Real-time scheduling: Fixed Priority Scheduling

Execute FPS schedule \( \not\approx \) Fails!

\[
U = \sum_{i=1}^{n} \frac{C_i}{T_i} = 1 > 0.779 \approx N(2^\frac{3}{2} - 1) \approx U_{\text{max}}
\]

Utilization test fails, schedulability not guaranteed.

Reduced task set

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

Utilization test fails, schedulability not guaranteed.

Execute FPS schedule \( \approx \) Works!

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

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

Further reduced task set

U = \sum_{i=1}^{n} \frac{C_i}{T_j} = \frac{4}{16} + \frac{3}{12} + \frac{1}{4} = 0.75 < 0.779 \approx N(2^{\frac{3}{5}} - 1) \equiv U_{\text{max}}

Utilization test succeeds, schedulability guaranteed.

Real-time scheduling: Fixed Priority Scheduling

Execute FPS schedule Works!

U = \sum_{i=1}^{n} \frac{C_i}{T_j} = \frac{4}{16} + \frac{3}{12} + \frac{1}{4} = 0.75 < 0.779 \approx N(2^{\frac{3}{5}} - 1) \equiv U_{\text{max}}

Utilization test succeeds, schedulability guaranteed.

Real-time scheduling: Fixed Priority Scheduling

Worst case response times

U = \sum_{i=1}^{n} \frac{C_i}{T_j} = \frac{4}{16} + \frac{3}{12} + \frac{1}{4} = 0.75 < 0.779 \approx N(2^{\frac{3}{5}} - 1) \equiv U_{\text{max}}

Utilization test succeeds, schedulability guaranteed.

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

Worst case response times

for others tasks: \( R_j = C_j + I_j \) (interference from higher priority tasks)

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

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

\( \text{Iterate the recurrent form until: } R_j^{t+1} = R_j^t \) 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!

\( \text{All possible release combinations in a full hyper-cycle need to be considered!} \)

- 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

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

Fixed-point equation

Recurrence form:

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

Iterate until: \( R_j^{t+1}(a) = R_j^t(a) \) or \( R_j^{t+1}(a) > a + D_j \)

Worst case response times

\[ R_3 = 1 \leq 4 \checkmark; \ R_2 = 4 \leq 12 \checkmark; \ R_1 = 10 \leq 16 \checkmark; \ \sum_{j=1}^{N} \frac{C_j}{T_j} \leq N(2^N - 1) \checkmark \]
Real-time scheduling: Earliest Deadline First

Worst case response times

\[ R_j = \max \{ R_j(a) - a \} \in A \quad \text{where} \quad A = \text{scm} (T_i) \]

Testing all combinations in a hyper-cycle:

\[ R_3 = 1 \leq 4 \checkmark; \quad R_2 = 4 \leq 12 \checkmark; \quad R_1 = 12 \leq 16 \checkmark; \quad \frac{\sum_{i=1}^{n} C_i}{T_j} > N(2^{\frac{n}{2}} - 1) \times \]

Real-time scheduling: Fixed Priority Scheduling

Worst case response times

\[ R_3 = 1 \leq 4 \checkmark; \quad R_2 = 4 \leq 12 \checkmark; \quad R_1 = 12 \leq 16 \checkmark; \quad \frac{\sum_{i=1}^{n} C_i}{T_j} > N(2^{\frac{n}{2}} - 1) \times \]

Real-time scheduling: Earliest Deadline First

Worst case response times

\[ R_3 = 4 \leq 4 \checkmark; \quad R_2 = 12 \leq 12 \checkmark; \quad R_1 = 16 \leq 16 \checkmark; \quad \frac{\sum_{i=1}^{n} C_i}{T_j} \leq 1 \checkmark \]
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: Comparison

Response Time Analysis

<table>
<thead>
<tr>
<th>Utilization Test</th>
<th>( \sum_{i=1}^{n} \frac{C_i}{T_i} \leq \frac{N}{2^N - 1} )</th>
<th>Utilization Test</th>
<th>( \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDF</td>
<td>( \checkmark )</td>
<td>( \checkmark )</td>
<td>( \checkmark )</td>
</tr>
<tr>
<td>FPS</td>
<td>( \nabla )</td>
<td>( \nabla )</td>
<td>( \nabla )</td>
</tr>
</tbody>
</table>

- 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
Scheduling

Real-World Extension

More Realistic 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

Sporadic and Aperiodic Processes

Hard real-time tasks
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.
Sporadic and Aperiodic Processes

**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.

**Sporadic task utilizing deferrable server**

- Schedule must also work with less interference.

- Deferrable server only deploying if there are requests from the sporadic / aperiodic task.

**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.

**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.
Scheduling

Sporadic and Aperiodic Processes

Introducing an EDF server

The EDF equivalent to a deferrable server: a periodic server task with an immediate deadline.

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.

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
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 $W$ violates DMPO.

2. Generate $W'$ by swapping $P_i$ and $P_j$ if $P_i < P_j$ and $(D_i > D_j)$ violates DMPO.

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 $P'_i > P'_j \Rightarrow R'_j \leq R_j \Rightarrow R'_j \leq D_j$.
   3c. $t_i$ is schedulable in $W'$ because:
      - $t_i$ interferes only once with $t_j$.
      - $t_j$ released once in $R_j$.

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:

\[
\text{in } W: R_i \leq D_j < D_i \leq T_j \Rightarrow R_j < T_i \forall t_j \text{ interfered only once with } t_i \\
\text{also: } t_i \text{ released once in } R_j \text{ and } R_j < R_i \\
\text{in } W':
\]

- 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 \( t_j \) is released only after the former release of \( t_j \) is completed.

- In case that \( R_j > T_j \) for a specific scheduling situation, the following release of task \( t_j \) is delayed by \( R_j - T_j \).
- Mind that \( R_j > T_j \) cannot hold for all release situations, otherwise the task is not schedulable.
- The worst case response time \( |R_j| \) might thus be longer than \( T_j \) but still be shorter than \( D_j \).
Scheduling

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 $t_j$ 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:

\[
R_i(q) = B_i + qC_i + \sum_{k \geq 1} \left[ \frac{R_i(q)}{k} \right] C_k \quad \text{where} \quad \forall q R_i(q) - (q - 1)T_i \leq D_i,
\]

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

\[
R_i = \max \{ R_i(q) - (q - 1)T_i \mid q \in \{1, 2, \ldots, q_{\text{max}}\} \} \quad \text{and} \quad q_{\text{max}} = \left\lfloor \frac{R_i(q)}{T_i} \right\rfloor \leq T_i
\]

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 $\implies$ DMPO or RMPO.

Schedulable under DMPO or RMPO.
Interdependent tasks

Task dependencies

Lock requests by two 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 $Q$.

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

**Without priority inheritance**

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

**Priority inversion**

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.

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 $C_k$:
  \[ C_k = \max \{ \text{employ}(i, k) \cdot P_i \} \]
  where $\text{employ}(i, k)$ is a boolean function returning "1" for true if task $t_i$ employs resource $R_k$.
- Each task $t_i$ has a dynamic priority $P_i^D$:
  \[ P_i^D = \max \{ P_i \cdot \text{locked}(i, k) \cdot C_k \} \]
  with $\text{locked}(i, k)$ being a boolean function returning "1" for true if task $t_i$ holds resource $R_k$. 

Circular task dependencies

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

**Ceiling Priority Protocol**

- **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.

**Maximal blocking time:**

\[ B_i = \max \{ \text{usage}(r, i) \cdot C(r) \}^R = 1 \]

- \( R \) 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 a **lower priority task**.

**Real-World Extension**

- **More Realistic 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
Non pre-emptive scheduling

In pre-emptive scheduling:
- Maximal individual blocking times $B_t$ can be determined for each task $t_i$ by employing a priority ceiling protocol.
- Maximum overall blocking time $B_{\text{max}} = \max(B_t)$.

Cooperative Scheduling

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

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 $B_{\text{max}}$ additional tasks need to be engineered to participate in a specific cooperative schedule.
- Requires that a value $B_{\text{max}}$ can be accepted by all tasks.
- Short and reactive tasks are excluded or treated separately.

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.
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 \( T_f \):

\[
R_i = B_i + C_i + \sum_{j > i} \left[ \frac{R_j}{T_j} \right] C_j + \max \left[ \frac{R_k}{T_k} \right] C_k^f \quad \text{for } k \geq i
\]

If error recovery is performed at the highest priority:

\[
R_i = B_i + C_i + \sum_{j > i} \left[ \frac{R_j}{T_j} \right] C_j + \max \left[ \frac{R_k}{T_k} \right] C_k^f
\]

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

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

**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 tasks (priority inheritance, priority ceiling protocols).
- Cooperative and deferred pre-emption scheduling.
- Fault tolerance in terms of exception handling considerations.

**Language support**
- Ada, POSIX