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.

References for this chapter

[Burns2007] Alan Burns and Andy Wellings
Concurrent and Real-Time Programming in Ada
Cambridge University Press (2007)

[Murthy2001] CSR Murthy, G Manimaran
Resource Management in Real-Time Systems and Networks
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

- **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).

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

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.

- 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)

© 2015 Uwe R. Zimmer, The Australian National University
**Real-time scheduling: Fixed Priority Scheduling**

**Task set**

- (4, 1)
- (12, 3)
- (16, 8)

**Execute FPS schedule \( \not\leadsto \) Fails!**

**Maximal utilization**

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

where \( C_i \) is the computation time and \( T_i \) is the length of the period for task \( i \) out of \( N \) tasks.

-\( \text{Sufficient, yet not necessary test} \)
Real-time scheduling: Fixed Priority Scheduling

Execute FPS schedule \( \checkmark \) Works!

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

\( \triangleright \) Utilization test fails, schedulability not guaranteed.

Reduced task set

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

Further reduced task set

\[ U = \sum_{j = 1}^{\pi} \frac{C_j}{T_j} = \frac{4}{16} + \frac{3}{12} + \frac{1}{4} = 0.75 < 0.779 \approx N(2^{\frac{3}{2}} - 1) = U_{\text{max}} \]

\( U \) utilization test succeeds, schedulability guaranteed.

Execute FPS schedule \( \Rightarrow \) Works!

Real-time scheduling: Fixed Priority Scheduling

Worst case response times

\[ U = \sum_{j = 1}^{\pi} \frac{C_j}{T_j} = \frac{4}{16} + \frac{3}{12} + \frac{1}{4} = 0.75 < 0.779 \approx N(2^{\frac{3}{2}} - 1) = U_{\text{max}} \]

\( U \) utilization test succeeds, schedulability guaranteed.

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

### Worst case response times

$$ R_j = C_j + I_j \quad \text{(interference from higher priority tasks)} $$

for others tasks:

$$ R_j = C_j + \sum_{k > j} \left\lfloor \frac{R_j}{T_k} \right\rfloor \cdot C_k $$

Real-time scheduling: Earliest Deadline First

### Response time analysis

$$ R_j = C_j + \sum_{k > j} \left\lfloor \frac{R_j}{T_k} \right\rfloor \cdot C_k $$

Fixed-point equation

Recurrent form:

$$ R_j^{t+1} = C_j + \sum_{k > j} \left\lfloor \frac{R_j^t}{T_k} \right\rfloor \cdot C_k \quad \text{with:} \quad R_j^0 = C_j $$

Iterate the recurrent form until:

$$ R_j^{t+1} = R_j^t \quad \text{or} \quad R_j^{t+1} > D_j $$

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.
Scheduling

Real-time scheduling: Earliest Deadline First

Response time analysis

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

Iterate until:

\[ R_j^{t+1}(a) = R_j^t(a) \]

for

\[ R_j^0 = a + C_j \]

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

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

Real-time scheduling: Fixed Priority Scheduling

Worst case response times

\[ R_3 = 1 \leq 4 \checkmark; \ R_2 = 4 \leq 12 \checkmark; \ R_1 = 19 > 16 \times; \ \sum_{j=1}^{n} \frac{C_j}{T_j} > N(2^{\frac{1}{N}} - 1) \times \]

Testing all combinations in a hyper-cycle:

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

**Worst case response times**

\[ R_3 = 1 \leq 4 \checkmark; R_2 = 8 \leq 12 \checkmark; R_1 = 12 \leq 16 \checkmark; \sum_{j=1}^{n} \frac{C_j}{T_j} \leq 1 \checkmark \]

Scheduling

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>Utilization Test</td>
</tr>
<tr>
<td>( \sum_{j=1}^{n} \frac{C_j}{T_j} \leq N(2^N - 1) )</td>
<td>( \sum_{j=1}^{n} \frac{C_j}{T_j} \leq 1 )</td>
</tr>
<tr>
<td>Response Times (( R_j ))</td>
<td>Response Times (( R_j ))</td>
</tr>
<tr>
<td>((T_6, C_6) = ((16, 6), (12, 3), (4, 1)))</td>
<td>(\times)</td>
</tr>
<tr>
<td>((T_5, C_5) = ((16, 6), (12, 3), (4, 1)))</td>
<td>(\times)</td>
</tr>
<tr>
<td>((T_4, C_4) = ((16, 4), (12, 3), (4, 1)))</td>
<td>(\checkmark)</td>
</tr>
</tbody>
</table>

Fixed Priority Scheduling \(\leftrightarrow\) 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
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

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
Scheduling

Sporadic and Aperiodic Processes

FPS for hard real-time tasks

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

Response times for sporadic / aperiodic task can be large.

FPS lowest priority for soft real-time task

Sporadic / aperiodic task set to lowest priority.

Introducing a soft real-time task

Introducing a server task on highest priority

Setting a deferrable server task as a proxy for sporadic / aperiodic tasks on highest priority level.
Sporadic and Aperiodic Processes

FSP server task as normal task

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

FSP task as normal task

- Schedule must also work with less interference.

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

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.

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.

Earliest Deadline Last (EDL) for sporadic tasks

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

G 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
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 \( W \) \( \not\in DMPO \)
2. Generate \( W' \) by swapping \( P_i \) and \( P_j \) \( \in (P_i < P_j') \land (D_i > D_j') \not\in DMPO \)
3. \( W' \) schedules \( Q \) because:
   3a. All \( t_k \in Q \) with \( P_k > P_i \) or \( P_k < P_j \) are unaffected.
   3b. \( t_i \) is schedulable in \( W' \) because \( P_i' > P_j' \Rightarrow R_i' \leq R_j \leq D_j \Rightarrow R_j' \leq D_j \)
   3c. \( t_j \) is schedulable in \( W' \) because:
      in \( W \): \( R_j \leq D_j < D_j \leq T_j \Rightarrow R_j < T_j \), meaning that \( t_j \) interfered only once with \( t_j \)
      also: \( t_j \) released once in \( R_j \) and \( R_i < R_j \)
      \( \Rightarrow \) in \( W' \): \( t_j \) interferes only once with \( t_j \) if \( R_i' = R_j \leq D_j < D_j \Rightarrow R_i' < D_j \).
Proof of DMPO optimality

Swap two priorities out of W which violate DMPO:

\[ \begin{align*}
W &:= W', \\
\text{in } W: & R_i \leq D_j < D_j \leq T_j \Rightarrow R_j < T_j \Rightarrow t_j \text{ interfered only once with } t_i \\
\end{align*} \]

Also: \( t_j \) released once in \( R_j \) and \( R_j < R_i \)

In \( W' \):

\[ \begin{align*}
\end{align*} \]

\( \text{Deadline monotonic ordering is optimal:} \)

\( \text{(if a process set is schedulable under an FPS-scheme, then it is also schedulable under FPS with deadline monotonic priorities.)} \)
Interdependent tasks

Independent tasks

Deadlines identical to cycle times \( R_{max} \) or \( R_{min} \).

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.

Since the response time \( R \) can now be potentially greater than the cycle time \( T \):

- more than one release \( q \) of the task \( t_j \) needs to be considered:

\[
R_j(q) = B_j + qC_j + \sum_{k \geq j} \frac{R_k(q)}{k} C_k \quad \text{where } \forall q [R_j(q) - (q - 1)T_j \leq D_j]
\]

\( B_j \) is the blocking time; \( q \) is the number of releases.

\[
R_j = \max \{R_j(q) - (q - 1)T_j | q \in \{1...q_{max}\} \} \quad \text{and } q_{max} = \left\lfloor \frac{R_j(q)}{T_j} \right\rfloor + 1
\]

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
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 $Q$.

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.

Without priority inheritance

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

Task dependencies with multiple locks

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

#### Interdependent tasks

**Task dependencies with multiple locks**

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

**Circular task dependencies**

Scheduling by DMPO/RMPO results in deadlock.

(Priority inheritance does not make a difference for blocked 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 $\text{employ}(i, k)$ being a boolean function returning “1” for true if task $t_i$ employs resource $R_k$.
- Each task $t_i$ has a **dynamic priority** $P_i^D$:
  \[
  P_i^D = \max \{ P_i \max \{ \text{locked}(i, k) \cdot C_k \k \}
  \]
  with $\text{locked}(i, k)$ being a boolean function returning “1” for true if task $t_i$ holds resource $R_k$. 
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: \( B_i = \max \{usage(r,i) \cdot C(r)\}_i \)

- \( R \) denoting the number of critical sections.
- \( usage(r,i) \) 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
  - Deadlines are identical with task’s period time \( (D = T) \)
  - Tasks are independent
  - Pre-emptive scheduling
  - Worst case execution times are known

we will introduce sporadic and aperiodic processes

we will introduce arbitrary deadlines

we will introduce schedules for interacting tasks

we will introduce (briefly) cooperative scheduling

we will introduce fault tolerant scheduling
Non-preemptive scheduling

Cooperative Scheduling

Every task \( t_i \) is divided into \( k \) non-preemptive 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.

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.

Response times:

\[
R_j = R^n_i + F_j \quad \text{with} \quad R^{k+1}_i = B_{\text{max}} + C_i - F_j + \sum_{j>i} \left[ \frac{R^k_j}{T_j} \right] C_j 
\]

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

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

\[
R_j = R^n_i \quad \text{with} \quad R^{k+1}_i = C + \sum_{j>i} \left[ \frac{R^k_j}{T_j} \right] C
\]

For the further simplified case of \( \forall \leq T = T_j \):

\[
R_j = C + \sum_{j>i} C
\]
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

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 \(T_f\):

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

If error recovery is performed at the highest priority:

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

General scheduling methods

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

- Without any further restrictions this problem is \(NP\)-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