Motivation

Scheduling in Real-Time Systems

Concurrence 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-time:** 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 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

Earliest Deadline First (EDF)

1. Determine one of the process(es) with the earliest deadline.
2. Execute this process
   2a. until another process deadline is fixed earlier than the current one.
   2b. then the current one.

Times to deadlines

Gives an idea how “critical” the schedule is.

Worst case response times

Response time: Time from scheduler request to process completion

Maximal utilization

If deadlines $D_i$ are identical to cycle times $T_i$ for each task $i$:
The maximal utilization for EDF becomes $\sum \frac{T_i}{D_i} \leq 1$ (sufficient and necessary test).

Fixed Priority Scheduling (FPS), rate monotonic

1. Each process is assigned a fixed priority according to its cycle time:
   $T_T < T_T \Rightarrow P \geq P$
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 offline.
   - 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.
Scheduling

Real-time scheduling: Fixed Priority Scheduling

Execute FPS schedule \(\Rightarrow\) Fails!

Maximal utilization

\[
U = \frac{\sum C_i}{\sum T_i} = \frac{6}{10} + \frac{3}{12} + \frac{1}{4} = 0.875 > 0.779 = N(2^3 - 1) = U_{\text{max}}
\]

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

Scheduling

Real-time scheduling: Fixed Priority Scheduling

Reduced task set

Execute FPS schedule \(\Rightarrow\) Works!

Maximal utilization

\[
U = \frac{\sum C_i}{\sum T_i} = \frac{6}{10} + \frac{3}{12} + \frac{1}{4} = 0.875 > 0.779 = N(2^3 - 1) = U_{\text{max}}
\]

\(\Rightarrow\) Utilization test succeeds, schedulability guaranteed.

Scheduling

Real-time scheduling: Fixed Priority Scheduling

Further reduced task set

Execute FPS schedule \(\Rightarrow\) Works!

Maximal utilization

\[
U = \frac{\sum C_i}{\sum T_i} = \frac{4}{10} + \frac{1}{2} + \frac{1}{4} = 0.75 < 0.779 = N(2^3 - 1) = U_{\text{max}}
\]

\(\Rightarrow\) Utilization test succeeds, schedulability guaranteed.

© 2019 Uwe R. Zimmer, The Australian National University page 754
Real-time scheduling: Fixed Priority Scheduling

Worst case response times

for the highest priority task: $R_1 = C_3$

for others tasks: $R_i = C_i + T_i$ (interference from higher priority tasks)

with:

$$ R_j = \frac{a + T_j - T_k}{T_j} + 1 \cdot C_k $$

$$ R_i = C_i + \sum_{k \in A \cap \{i\}} \left( \frac{a + T_j - T_k}{T_j} + 1 \right) \cdot C_k $$

for the highest priority task: $R_1 = C_3$

for others tasks: $R_i = C_i + \sum_{k \in A \cap \{i\}} \left( \frac{R_j}{T_j} \right) \cdot C_k$

Response time analysis

$R_j(a) = \frac{a}{T_j} + 1 \cdot C_j + \sum_{k \in A \cap \{j\}} \left( \frac{R_k(a)}{T_k} \right) \cdot C_k$

Recurrence form: $R_j^{k+1} = \frac{a}{T_j} + 1 \cdot C_j + \sum_{k \in A \cap \{j\}} \left( \frac{R_k^{k+1}}{T_k} \right) \cdot C_k$

Response time analysis

$R_j(a) = \frac{a}{T_j} + 1 \cdot C_j + \sum_{k \in A \cap \{j\}} \left( \frac{R_k(a)}{T_k} \right) \cdot C_k$

Recurrence form: $R_j^{k+1} = \frac{a}{T_j} + 1 \cdot C_j + \sum_{k \in A \cap \{j\}} \left( \frac{R_k^{k+1}}{T_k} \right) \cdot C_k$

Iterate until: $R_j^{k+1} > D_j$ or $R_j^{k+1} = R_j^k$
Real-time scheduling: Fixed Priority Scheduling

Worst case response times

\[ R_3 = 1 \leq 4 \land R_2 = 4 \leq 12 \land R_1 = 12 \leq 16 \land \sum_{i=1}^{3} C_i \leq N(2^3 - 1) \]

Real-time scheduling: Earliest Deadline First

Worst case response times

\[ R_3 = 1 \leq 4 \land R_2 = 4 \leq 12 \land R_1 = 19 \leq 16 \land \sum_{i=1}^{3} C_i \leq N(2^3 - 1) \]

Testing all combinations in a hyper-cycle:

\[ R = \min (R_i(a) - T_{i,k} + A) \quad \text{where} \quad A = A^{BCB} (T_i) \]

Real-time scheduling: Comparison

Response Time Analysis

Fixed Priority Scheduling ↔ Earliest Deadline First

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

- Graceful degradation (no source is over-bounded):
  - FPS: processes with lower priorities will always miss their deadlines first.
  - EDF: any process can miss its deadline and can trigger a cascade of missed deadlines.

- Response time analysis and utilization tests:
  - FPS: On-line utilization test ↔ response time analysis: fixed point equation
  - EDS: On-line utilization test ↔ response time analysis: fixed point equation in hyper-cycle

- Tasks are periodic
- Deadlines are identical with task’s period time \(D - 1\)
- Tasks are independent
- Pre-emptive scheduling
- Worst case execution times are known
**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**

- 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

**FPS for hard real-time tasks**

**Introducing a soft real-time task**

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

**Introducing a server task on highest priority**

**FPS server task as normal task**

**Sporadic task utilizing deferrable server**

© 2019 Uwe R. Zimmer, The Australian National University
Sporadic and Aperiodic Processes

Sporadic task utilizing sporadic server

As sporadic servers only replenish after a fixed time after their actual deployment.

- Prevents load loss or overload to a deferrable server.
- Minimal interarrival time knowledge is employed.

Earliest Deadline Last (EDL) for sporadic tasks

Earliest Deadline Last (EDL) scheduling (while still keeping all deadlines)

- when sporadic or aperiodic tasks are scheduled.
- Deadlines explicitly pushed to their limits during the EDL phase.

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. $t_y, t_x$ are two tasks in $Q$ with $P_j > P_l$ and $D_j > D_l$ in $W$ so DMPO
2. Generate $W'$ by swapping $P_j$ and $P_l$ so $P_j < P_l$ and $D_j > D_l$ so DMPO
3. $W'$ schedules $Q$ because:
   3a. All $t_y \in Q$ with $P_j > P_l$ or $P_j < P_l$ are unscheduled.
   3b. $t_y$ is schedulable in $W'$ because $P_j > P_l$ and $D_j > D_l$.
   3c. $t_j$ is schedulable in $W'$ because $P_j > P_l$ and $D_j > D_l$.

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

Earliest Deadline Last (EDL) for sporadic tasks

Earliest Deadline Last (EDL) scheduling (while still keeping all deadlines)

- when sporadic or aperiodic tasks are scheduled.
- Deadlines explicitly pushed to their limits during the EDL phase.

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. $t_y, t_x$ are two tasks in $Q$ with $P_j > P_l$ and $D_j > D_l$ in $W$ so DMPO
2. Generate $W'$ by swapping $P_j$ and $P_l$ so $P_j < P_l$ and $D_j > D_l$ so DMPO
3. $W'$ schedules $Q$ because:
   3a. All $t_y \in Q$ with $P_j > P_l$ or $P_j < P_l$ are unscheduled.
   3b. $t_y$ is schedulable in $W'$ because $P_j > P_l$ and $D_j > D_l$.
   3c. $t_j$ is schedulable in $W'$ because $P_j > P_l$ and $D_j > D_l$.

Real-World Extension

we More Realistic Assumptions

- Tasks are periodic
- Deadlines are identical (as we will introduce sporadic and aperiodic processes)
- Tasks are independent
- Pre-emptive scheduling
- Worst case execution times are known
- We will introduce fault tolerant scheduling
Scheduling

Proof of DMPO optimality

Swap two priorities out of \( W \) which violate DMPO:

\[ W \]

in \( W \): \( D_1 < D_2 < D_3 < T_i = R_j < T_j \) or \( I_i \) interfered only once with \( I_j \)

also \( I_j \) released once in \( R_j \) and \( R_i < R_j \)

\[ W' \]

Proof of DMPO optimality

\[ \text{Swap two priorities out of } W \text{ which violate DMPO:} \]

in \( W \): \( D_1 < D_2 < D_3 < T_i = R_j < T_j \) or \( I_i \) interfered only once with \( I_j \)

also \( I_j \) released once in \( R_j \) and \( R_i < R_j \)

\[ W' \]

Scheduling

Swap all priorities out of \( W \) which violate DMPO:

as Swap all \( T_j \) in \( W \) with \( (P_j > P_i) \land (D_j > D_i) \) in \( W \)

resulting in all \( T_j \) in \( W \) with \( P_j > P_i \) to have \( D_j < D_i \)

Constituting the DMPO scheme

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

\[ \text{Deadline monotonic ordering is optimal:} \]

if a process \( i \) is schedulable under an FPS-scheme, then it is also schedulable under FPS with deadline monotonic priority.

- 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

\[ \text{More Realistic Assumptions} \]

- 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

Tasks with arbitrary deadlines

- \( D > T \) (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 \( R_j > T_i \) for specific scheduling situation, the following release of task \( j \) is delayed by \( k_T \)
- Mind that \( R_j > T_i \) cannot hold for all release situations, otherwise the task is not schedulable.

The worst case response time \( [R_j] \) might be longer than \( T_i \) but must still be shorter than \( D_j \)

- Deadline monotonic ordering is optimal:

  if a process \( i \) is schedulable under an FPS-scheme, then it is also schedulable under FPS with deadline monotonic priority.

\[ \text{Real-World Extension} \]

- 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

Tasks with arbitrary deadlines

- \( D > T \) (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 \( R_j \) it can now be potentially greater than the cycle time \( T \)

more than one release of the task \( j \) needs to be considered:

- Deadlines identical to cycle times \( L = T \)
- Pre-emptive scheduling

Worst case execution times are known

- \( R_j \) is the blocking time; \( q \) is the number of releases

\[ R_j = \sum_{q=1}^{R_j} \left[ T_j - L - T_i \right] \]

\[ q \leq 1 \] 
Interdependent tasks

Task dependencies

Lock requests by two tasks.

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

- $R$ denoting the number of critical sections.
- $\text{usage}(r, l)$ 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 blocks the higher priority task. (note that the blue task is unaffected.)

Priority inversion

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

Task dependencies with multiple locks

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

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

Circular task dependencies

Ceiling Priority Protocol

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

- Each task $t_j$ has a static priority $P_{r_j}$.
- Each resource $R_k$ has a static ceiling priority $C_{r_k}$:
  \[ C_{r_k} = \max \{ \text{employ}(i, k) \cdot P_{r_i} \} \]
  with $\text{employ}(i, k)$ being a boolean function returning “1” for true if task $i$ employs resource $R_k$.
- Each task $t_j$ has a dynamic priority $P^D_{r_j}$:
  \[ P^D_{r_j} = \max \{ P_{r_{\max}} \cdot \text{locked}(i, k) \cdot C_{r_k} \} \]
  with $\text{locked}(i, k)$ being a boolean function returning “1” for true if task $i$ holds resource $R_k$.

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

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

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

Considerations:
- Code block division need to be done thoroughly.
- Additional protection against misbehaving (non-cooperative) tasks: Scheduler pre-empt tasks if deferred pre-emption
- 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.

Fault Tolerance

Exceptions and Recoveries

Task $T_j$ needs extra CPU-time $C_j^f$ for error recovery or exception handling and the minimum inter-arrival time between faults $T_F$:

$$R_j = R_j^f + C_j + \sum_{i \neq j} R_i^f + C_i + \max \left\{ \frac{R_i^f}{T_i} | C_i \right\} R_i \geq i$$

If error recovery is performed at the highest priority:

$$R_j = R_j^f + C_j + \sum_{i \neq j} R_i^f + C_i + \max \left\{ \frac{R_i^f}{T_i} | C_i \right\}$$

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

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

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

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 (RMP) and Deadline Monotonic Priority Ordering (DMPO).
  - Earliest Deadline First (EDF).

- Real-world extension
  - 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