Scheduling

A scheduling scheme provides two features:

- Ordering the use of resources (e.g., CPUs, networks)
- 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 schemes

- Static: all predictions and schedules are done off-line
  - often better predictability = most hard real-time systems
- Dynamic: run-time situation is taken into account
  - more flexible, more efficient = most soft real-time systems

Real-time scheduling as task queuing

- 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 & Embedded Systems

Dynamic scheduling

Earliest deadline first (EDF)

1. Determine (one of) the process(es) with the closest deadline.
2. Execute this process
   2-a) until it finishes
   2-b) or until another process’ deadline is found closer than the current one.

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

Rate monotonic priorities

assign task priorities according to the cycle times $T_j$ (identical to deadline $D_j$).

Avoid task switches (in case of equal deadlines)

1. Schedule the earliest deadline first
2. Avoid task switches (in case of equal deadlines)

Earliest deadline first: Response times

worst case response times $R_i$ (maximal time in which the request from task $T_i$ is served).

Earliest deadline first: Maximal utilization

maximal possible utilization $\sum \frac{C_j}{T_j} \leq 1$ = sufficient & necessary test!

with $C_j / T_j$ the computation and cycle times of task $j$.

(The deadlines $D_j$ are assumed to be identical with the cycles $T_j$ here.)

Real-Time & Embedded Systems

Static scheduling: Fixed Priority Scheduling (FPS), rate monotonic

1. Each process is assigned a fixed priority according to its cycle time $T_j$:
   $T_j < T_j' \Rightarrow P_i > P_j'$
2. At any point in time: dispatch the process with the highest priority

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

- Rate monotonic ordering is optimal (in the framework of fixed priority schedulers), i.e. if a process set is schedulable under a FPS-scheme, it is also schedulable by applying rate monotonic priorities.

Earliest deadline first

Since the dispatched process is selected at run-time, due to the current deadlines.

Real-Time & Embedded Systems

Example: Deadlines

Real-Time & Embedded Systems

Dynamic scheduling: Earliest Deadline First (EDF)

Earliest deadline first: Response times

worst case response times $R_i$ (maximal time in which the request from task $T_i$ is served):

- can be close or identical to deadlines.
- small or none spare capacity, if any task misses its expected computation time.

Real-Time & Embedded Systems

Static scheduling

Fixed Priority Scheduling (FPS), rate monotonic

1. Each process is assigned a fixed priority according to its cycle time $T_j$:
   $T_j < T_j' \Rightarrow P_i > P_j'$
2. At any point in time: dispatch the process with the highest priority

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

- Rate monotonic ordering is optimal (in the framework of fixed priority schedulers), i.e. if a process set is schedulable under a FPS-scheme, it is also schedulable by applying rate monotonic priorities.
Real-Time & Embedded Systems

Static scheduling: Fixed Priority Scheduling (FPS), rate monotonic

Rate monotonic priorities

max. utilization test: \( \sum_{i=1}^{n} \frac{C_i}{T_i} \leq \frac{\frac{1}{N}}{N-1} \)

⇒ assign task priorities according to the cycle times \( T_i \) (identical to deadline \( D_i \)).

Max utilization:

\[
\text{utilization test: } \sum_{i=1}^{n} \frac{C_i}{T_i} \leq \frac{\frac{1}{N}}{N-1} \Rightarrow \text{sufficient, but not necessary test!}
\]

\[
\text{utilization test: } \sum_{i=1}^{n} \frac{C_i}{T_i} > \frac{0.779}{3} \Rightarrow \text{not guaranteed!}
\]

Rate monotonic priorities (reduced requests)

max. utilization test: \( \sum_{i=1}^{n} \frac{C_i}{T_i} \leq \frac{\frac{1}{N}}{N-1} \)

⇒ utilization: \( \frac{6}{16} + \frac{3}{12} + \frac{1}{4} = 0.875 > 0.779 = \frac{0.779}{3} \Rightarrow \text{not guaranteed!} \)

Rate monotonic priorities (further reduced requests)

max. utilization test: \( \sum_{i=1}^{n} \frac{C_i}{T_i} \leq \frac{\frac{1}{N}}{N-1} \)

⇒ utilization: \( \frac{4}{16} + \frac{1}{12} + \frac{1}{4} = 0.75 > 0.779 = \frac{0.779}{3} \Rightarrow \text{not guaranteed!} \)

Static scheduling: Fixed Priority Scheduling (FPS), rate monotonic

Rate monotonic priorities

usage test: \( \sum_{i=1}^{n} \frac{C_i}{T_i} > \frac{0.779}{3} \Rightarrow \text{not guaranteed!} \)
Utilisation based Analysis for FPS rate monotonic

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

- with \( C_i \), the computation time and \( T_i \), the length of the period for task \( i \) out of \( N \) tasks and assuming that the deadline \( D_i = T_i \)
- sufficient, but not necessary
- \( O(n) \) complexity

\[ \Rightarrow \text{calculate the worst case response times for each task individually.} \]

For other tasks: \( R_i = C_i + I_j = \text{computation } C_i + \text{interference } I_j \)

\[ R_i^{k+1} = C_i + \sum_{j=1}^{n} \frac{R_j^k}{T_j} C_j \]

Example (further reduced requests):
- \( k = 1 \)
- \( C_i \) at priorities \( 1:2:3 \)
- \( R_i^0 = 1 (\ast) \)

\[ R_i^{k+1} = C_i + \sum_{j=1}^{n} \frac{R_j^k}{T_j} C_j \]

Example (further reduced requests):
- \( k = 1 \)
- \( C_i \) at priorities \( 1:2:3 \)
- \( R_i^0 = 1 (\ast) \)

\[ R_i^{k+1} = C_i + \sum_{j=1}^{n} \frac{R_j^k}{T_j} C_j \]

Example (further reduced requests):
- \( k = 1 \)
- \( C_i \) at priorities \( 1:2:3 \)
- \( R_i^0 = 1 (\ast) \)

\[ R_i^{k+1} = C_i + \sum_{j=1}^{n} \frac{R_j^k}{T_j} C_j \]

Example (further reduced requests):
- \( k = 1 \)
- \( C_i \) at priorities \( 1:2:3 \)
- \( R_i^0 = 1 (\ast) \)
Response time analysis

\[ R_i^{k+1} = C_i + \sum_{j=1}^{k} \left( \frac{R_{ij}}{T_j} \right) C_j \quad \text{where} \quad R_i^0 = C_i \]

- \( R_i^k \) is the system state
- \( C_i \) is the workload
- \( R_{ij} \) is the service time of task \( j \)
- \( T_j \) is the cycle time of task \( j \)

Example (reduced requests):
- set of tasks \( \{(T_1, C_1)\} = \{(16,6):(12,3):(4,1)\} \) at priorities \( \{1:2:3\} \), \( R_i^0 = 6 \)
- \( R_1^1 = 6 + \left( \frac{6}{12} \right)3 + \left( \frac{4}{4} \right)1 = 11 \)
- \( R_1^2 = 6 + \left( \frac{11}{12} \right)3 + \left( \frac{4}{4} \right)1 = 12 \)
- \( R_1^3 = 6 + \left( \frac{12}{12} \right)3 + \left( \frac{12}{4} \right)1 = 12(\nu) \)

Response time analysis (full requests):
- set of tasks \( \{(T_1, C_1)\} = \{(16,8):(12,3):(4,1)\} \) at priorities \( \{1:2:3\} \), \( R_i^0 = 8 \)
- \( R_1^1 = 8 + \left( \frac{8}{12} \right)3 + \left( \frac{3}{4} \right)1 = 13 \)
- \( R_1^2 = 8 + \left( \frac{12}{12} \right)3 + \left( \frac{4}{4} \right)1 = 18(\times) \)
- \( R_1^3 = 8 + \left( \frac{18}{12} \right)3 + \left( \frac{18}{4} \right)1 = 19(\times) \)

Response time analysis (full requests):
- set of tasks \( \{(T_1, C_1)\} = \{(16,6):(12,3):(4,1)\} \) at priorities \( \{1:2:3\} \), \( R_i^0 = 6 \)
- \( R_1^1 = 6 + \left( \frac{6}{12} \right)3 + \left( \frac{4}{4} \right)1 = 11 \)
- \( R_1^2 = 6 + \left( \frac{11}{12} \right)3 + \left( \frac{4}{4} \right)1 = 12 \)
- \( R_1^3 = 6 + \left( \frac{12}{12} \right)3 + \left( \frac{12}{4} \right)1 = 12(\nu) \)

Response time analysis (full requests):
- set of tasks \( \{(T_1, C_1)\} = \{(16,8):(12,3):(4,1)\} \) at priorities \( \{1:2:3\} \), \( R_i^0 = 8 \)
- \( R_1^1 = 8 + \left( \frac{8}{12} \right)3 + \left( \frac{3}{4} \right)1 = 13 \)
- \( R_1^2 = 8 + \left( \frac{12}{12} \right)3 + \left( \frac{4}{4} \right)1 = 18(\times) \)
- \( R_1^3 = 8 + \left( \frac{18}{12} \right)3 + \left( \frac{18}{4} \right)1 = 19(\times) \)

Example (reduced requests):
- set of tasks \( \{(T_1, C_1)\} = \{(16,6):(12,3):(4,1)\} \) at priorities \( \{1:2:3\} \), \( R_i^0 = 6 \)
- \( R_1^1 = 6 + \left( \frac{6}{12} \right)3 + \left( \frac{4}{4} \right)1 = 11 \)
- \( R_1^2 = 6 + \left( \frac{11}{12} \right)3 + \left( \frac{4}{4} \right)1 = 12 \)
- \( R_1^3 = 6 + \left( \frac{12}{12} \right)3 + \left( \frac{12}{4} \right)1 = 12(\nu) \)

Example (full requests):
- set of tasks \( \{(T_1, C_1)\} = \{(16,8):(12,3):(4,1)\} \) at priorities \( \{1:2:3\} \), \( R_i^0 = 8 \)
- \( R_1^1 = 8 + \left( \frac{8}{12} \right)3 + \left( \frac{3}{4} \right)1 = 13 \)
- \( R_1^2 = 8 + \left( \frac{12}{12} \right)3 + \left( \frac{4}{4} \right)1 = 18(\times) \)
- \( R_1^3 = 8 + \left( \frac{18}{12} \right)3 + \left( \frac{18}{4} \right)1 = 19(\times) \)

Response time analysis (full requests):
- set of tasks \( \{(T_1, C_1)\} = \{(16,6):(12,3):(4,1)\} \) at priorities \( \{1:2:3\} \), \( R_i^0 = 6 \)
- \( R_1^1 = 6 + \left( \frac{6}{12} \right)3 + \left( \frac{4}{4} \right)1 = 11 \)
- \( R_1^2 = 6 + \left( \frac{11}{12} \right)3 + \left( \frac{4}{4} \right)1 = 12 \)
- \( R_1^3 = 6 + \left( \frac{12}{12} \right)3 + \left( \frac{12}{4} \right)1 = 12(\nu) \)

Response time analysis (full requests):
- set of tasks \( \{(T_1, C_1)\} = \{(16,8):(12,3):(4,1)\} \) at priorities \( \{1:2:3\} \), \( R_i^0 = 8 \)
- \( R_1^1 = 8 + \left( \frac{8}{12} \right)3 + \left( \frac{3}{4} \right)1 = 13 \)
- \( R_1^2 = 8 + \left( \frac{12}{12} \right)3 + \left( \frac{4}{4} \right)1 = 18(\times) \)
- \( R_1^3 = 8 + \left( \frac{18}{12} \right)3 + \left( \frac{18}{4} \right)1 = 19(\times) \)
we will introduce schedules for interacting tasks

- tasks are independent
- worst case execution times are known
- tasks are periodic
- pre-emptive scheduling
- deadlines are identical with task’s period time ($D = T$)
- we will introduce arbitrary deadlines
- pre-emptive scheduling
- we will introduce (briefly) cooperative scheduling
- worst case execution times are known
- we will introduce fault tolerant scheduling

Extensions which we will introduce:

- aperiodic, sporadic & ‘soft’ real-time tasks
Introducing soft real-time tasks

Introducing a server task: Sporadic Server

- Set can be scheduled using average computation and period times.
- Hard real-time tasks can be scheduled under worst case conditions (including worst case behaviours of soft real-time tasks).

Introducing soft real-time tasks

Introducing a server task: Deferrable Server

- Deferrable Server (DS): Capacity replenished every $T_i$ (here: 8).

Introducing a server task: Sporadic Server

- Sporadic Server (SS): Capacity replenished $T_i$ units after $T_i = \text{POSIX}$.

Introducing dual priorities

Introducing a server task to EDF

- Start hard rt-tasks in low priorities; promote them in time to higher ones.
Deadline Monotonic Priority Ordering (DMPO)

Any task set Q which is schedulable by a FPS scheme W, is also schedulable by DMPO!

Proof:
1. \( t_i, t_j \) are two tasks in Q, with \( P_i > P_j \) and \( D_j > D_i \) in W → DMPO
2. Generate \( W' \) by swapping \( P_i \) and \( P_j \), i.e. \( (P'_j < P'_i) \land (D_j > D_i) \) → DMPO
3. \( W' \) is scheduling Q because:
   - 3-a all \( t_k \in Q \) with \( P_k > P_i \) or \( P_k < P_j \) are unaffected
   - 3-b \( t_j \) is schedulable in \( W' \) because \( P'_j > P_j \Rightarrow R'_j \leq R_j \leq D_j \)
   - 3-c \( t_j \) is schedulable in \( W' \) because
     - in W: \( R_j \leq D_j \), \( t_j \) interfered only once with \( t_j \)
     - in \( W' \): \( t_j \) interfered only once with \( t_j \)

\( W' \) is still schedulable in \( W' \) because:

\[ t_j \text{ is still schedulable in } W' \text{ because:} \]

Any task set Q which is schedulable by a FPS scheme W, is also schedulable by DMPO!

Including tasks with deadlines shorter than their cycle time

Proof:
1. \( \forall t_i, t_j \) are two tasks in Q, with \( P_i > P_j \) and \( D_j > D_i \) in W → DMPO
2. Generate \( W' \) by swapping \( P_i \) and \( P_j \), i.e. \( (P'_j < P'_i) \land (D_j > D_i) \) → DMPO
3. \( W' \) is scheduling Q because:
   - 3-a all \( t_k \in Q \) with \( P_k > P_i \) or \( P_k < P_j \) are unaffected
   - 3-b \( t_j \) is schedulable in \( W' \) because \( P'_j > P_j \Rightarrow R'_j \leq R_j \leq D_j \)
   - 3-c \( t_j \) is schedulable in \( W' \) because
     - in W: \( R_j \leq D_j \), \( t_j \) interfered only once with \( t_j \)
     - in \( W' \): \( t_j \) interfered only once with \( t_j \)

\( W' \) is still schedulable in \( W' \) because:

Real-Time & Embedded Systems

Static scheduling: Fixed Priority Scheduling (FPS), DMPO

Any task set Q which is schedulable by a FPS scheme W, is also schedulable by DMPO!

- Swap all \( t_i, t_j \) in Q, with \( P_i > P_j \) and \( D_j > D_i \) in W resulting in all \( t_i, t_j \) in Q, with \( P_i > P_j \), have \( D_j > D_i \)

= Deadline Monotonic Priority Ordering (DMPO)

- Since the each swapping operation keeps schedulability, the final priority scheme (DMPO) is also schedulable.

= If FPS-DMPO is not schedulable, there is no schedulable FPS-scheme.
Scheduling: Interdependencies

Schedule for independent tasks

1. Priority inheritance
   - Task $t_j$ inherits the priority of $t_i$, if:
     1. $P_i < P_j$
     2. $t_j$ has locked a resource $Q$
     3. $t_j$ is blocked waiting for resource $Q$ to be released
   - Maximal blocking time for task $t_i$: $B_i = \sum_{r=1}^{R} \sum_{i=1}^{r} usage(r, i)C(r)$
     - with $R$ the number of critical sections
     - $usage(r, i)$ a boolean function indicating that $r$ is used by at least one $t_j$ with $P_j < P_i$ and at least one $t_j$ with $P_j \geq P_i$
     - $C(r)$ is the worst case computation time in critical section $r$
   - A task can only be blocked once for each employed resource!

Without priority inheritance

A more complex example

© 2004 Uwe R. Zimmer, The Australian National University
Deadlock

Meritence

Immediate ceiling priority protocol \( \text{POSSX, Ada, RT-Java} \)

- Each task \( t_i \) has static default priority \( P_{t_i} \).
- Each resource (lock, monitor) \( R_k \) has a static ceiling priority \( C_k \), which is the maximum of priorities of the tasks \( t_j \) which employ this resource.

\[
C_k = \max \{ \text{employ}(i, k) \cdot P_j \}
\]

- Each task \( t_i \) has a dynamic priority \( P_{t_i}^k \), which is the maximum of its own static priority and the ceiling priorities of any resource it has locked.

\[
P_{t_i}^k = \max \{ P_{t_i} \cdot \max \{ \text{locked}(i, k) \cdot C_k \} \}
\]
very short and reactive tasks are excluded or will be treated specially.

\[ \sum \]

\[ B_{\text{max}} \]

• Requires that a value can be accepted by all tasks

\[ \forall T_{ii} \]

… and with even \( T_j \):

\[ C_i \]

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

Scheduling — real-world considerations

Considering non-pre-emptive scheduling

Deferred pre-emption – Cooperative Scheduling

Response times:

\[ R_i = R_i^0 + F_i \quad \text{with} \quad R_i^{k+1} = B_{\text{max}} + C_i - F_i + \sum_{j>i} \left( R_j^k + F_j \right) \]

and \( F_j \), the execution time of the final code-block

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

\[ R_i = R_i^0 + C_i = C + \sum_{j>i} C_j \]

\[ \text{and with even } T_j \text{,} \]

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

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

Real-Time & Embedded Systems

Scheduling: Interdependencies: Priority ceiling protocols

Deferred pre-emption – Cooperative Scheduling

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

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

• with \( r \) the number of critical sections

• \( \text{usage}(r, i) \) a boolean \( \land \) function indicating that \( r \) is used by

• at least one \( t_j \) with \( P_j < P_i \) and at least one \( t_j \) with \( P_j \geq P_i \)

• \( C(r) \) is the worst case computation time in critical section \( r \)

A task can only be blocked once by any lower priority task!

Defered pre-emption – Cooperative Scheduling

• 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

Cooperative Scheduling

Deferred pre-emption – Cooperative Scheduling

What’s the cost?

• Code block division need to be done very thoroughly.

• Additional protection against badly behaving (non-cooperative) tasks:

  • Scheduler pre-empts tasks, which fail to offer a ‘de-schedule’ themselves.

  • Due to a central \( B_{\text{max}} \) value, additional tasks need to be engineered to participate in a specific cooperative schedule.

  • Requires that a \( B_{\text{max}} \) value can be accepted by all tasks

  • very short and reactive tasks are excluded or will be treated specially.

Cooperative Scheduling

Deferred pre-emption – Cooperative Scheduling

Considering deadlines beyond the release period

Real-Time & Embedded Systems

Scheduling: Interdependencies: Priority ceiling protocols

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

Real-Time & Embedded Systems

Scheduling: Interdependencies: Priority ceiling protocols

Immediate ceiling priority protocol (POSIX, Ada, RT-Java)
Tasks with $D > T$

(Deadline later than inter-arrival period)

(a cross-over of a hard, periodic and a soft real-time task)

Assuming that a task $t_i$ is released only after a former release of $t_j$ is completed.

- In case that $R > T$ for a specific scheduling situation, the following release of task $t_j$ is delayed until completion of the former release.
- Mind that $R > T$ 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 $D_i$.

Real-Time & Embedded Systems

Fault Tolerance

Exceptions and Recoveries

Task $t_i$ needs extra CPU-time $C_i^j$ for error recovery or exception handling (done at $P_i$) and the minimum inter-arrival time between faults is $T_j$:

$$R_i = B_i + C_i + \sum_{j=1}^{\infty} \left\lfloor \frac{R_i}{T_j} \right\rfloor C_i^j + \max \left( \frac{R_i}{T_j} C_i^j \right)$$

if error recovery or exception handling is performed at the highest priority:

$$R_i = B_i + C_i + \sum_{j=1}^{\infty} \left\lfloor \frac{R_i}{T_j} \right\rfloor C_i^j + \max \left( \frac{R_i}{T_j} C_i^j \right)$$

Scheduling support in different real-time languages / environments

Ada95 provides:

- Task and interrupt priorities (static, dynamic, active)
- Task attributes
- Prioritized entry queues
- Priority ceiling locking (ICPP)
- Schedulers (at least FIFO within priorities (pre-emptive) is requested)

Ada95 does not provide:

- Earliest Deadline First (EDF)
- Sporadic servers (a Ada95-implementation of a sporadic server is on the course page)
- Direct task execution time measurements e.g. POSIX or VxWorks timers

Real-Time & Embedded Systems

Scheduling — real-world considerations

Scheduling

- Considering task sets with release offsets

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

  - without any further restrictions this problem is NP-hard!

  - by introducing further assumptions about the granularity of the period times and deadlines:

  - the schedulability analysis' complexity can be reduced to be realistic.

Considering

‘fault-tolerance’

(additional CPU-time for exception handling and recovery)

Language Support for Scheduling : Ada95

package System is
  subtype Any_Priority is Integer;
  subtype Priority is Any_Priority range Any_Priority'First .. Any_Priority'Last;
  type T is not System any Priority;
  type System is not System any Priority
end System;

package Ada.Dynamic_Priorities is
  procedure Set_Priority (Priority : in Any_Priority; T : in System.Any_Priority
  := Ada.Task_Identification.Current_Task);
  function Get_Priority (T : in System.Any_Priority
end Ada.Dynamic_Priorities;

Real-Time & Embedded Systems
Public HighResolutionTime getStart();   public void               setPeriod (RelativeTime period);   public void               setStart (HighResolutionTime start);
Summary

Scheduling

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

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

- **Language support**
  - Ada95, POSIX = static, off-line analysis mostly — RT-Java = on-line, dynamic scheduling