Scheduling Algorithm for multiprogramming in a Hard-Real-Time Environment

Authors: C. L. Liu and James W. Layland

Key Result: An optimum fixed priority scheduler possesses an upper bound to processor utilization which can be as low as 70% for large task sets. Further, full processor utilization can be achieved by dynamically assigning priorities on the basis of current deadlines

Assumptions:
A1> The requests for all tasks for which hard deadlines exist are periodic, with constant interval between requests.
A2> Deadlines consist of run-ability constraints only - i.e. each task must be completed before the next request for it occurs
A3> Tasks are independent of each other
A4> Execution time for each task is a constant

Notation:
- We use \([x]\) to denote the largest integer smaller than or equal to x
- We use \(|x|\) to denote the smallest integer larger than or equal to x
- We use \({a/b}\) to denote \((a/b) - [a/b]\), i.e., the fractional part of \(a/b\).

Definition: A scheduling algorithm is a set of rules that determine the task to be executed at a particular instant

Terminology:
A real-time system comprises of \(m\) tasks. Associated with each task number \(i\) we have
- a period \(T_i\) (deadline equals period)
- a execution time \(C_i\)

The request rate of a task is the reciprocal of its period.

Definition: The critical instant for a task is defined to be an instant at which a request for that task will have the largest response time (worst-case completion time).

Fixed Priority Scheduling

Priority Assignment: Tasks with higher request rates have higher priorities (Rate monotonic priority assignment)

Theorem 1: A critical instant for any task occurs whenever the task is requested simultaneously with requests for all higher priority tasks.
PROOF:
Assume that the tasks are ordered in the decreasing order of priority (task $m$ is the lowest priority task).
Consider a particular instance of task $m$ that occurs at $t_1$. Consider all requests from task $i$ that occur at $t_2$.

\[ \text{figure: Execution of task } i \text{ between requests for task } m \]
Delay in the completion of task $m$ is largest when $t_2$ coincides with $t_1$

End of proof

An Important Observation:
Say, we have two tasks to be scheduled where $T_1 < T_2$. There are only two possible priority assignments:

- Task 1 is higher priority. According to the above theorem:
  \[ (_{T_2/T_1_})C_1 + C_2 \leq T_2 \quad (1) \]
  \text{this condition is necessary but not sufficient to guarantee the feasibility of the priority assignment}

- Task 2 is higher priority:
  \[ C_1 + C_2 \leq T_1 \quad (2) \]

Since,
\[ (_{T_2/T_1_})C_1 + (_{T_2/T_1_})C_2 \leq (_{T_2/T_1_})T_1 \leq T_2 \]
(2) implies (1). In other words, whenever $T_1 < T_2$ and $C_1, C_2$ are such that the task schedule is feasible with task 2 at higher priority than task 1, it is also feasible with task 1 at higher priority than task 2, but the opposite is not true.

THEOREM 2: If a feasible assignment exists for some task set, the rate monotonic priority assignment is feasible for that task set.
PROOF: Follows from theorem 1 and the above observation.

Achievable Processor Utilization (fixed priority scheduling)
Definition:

- Processor utilization factor is defined to be the fraction of the time spent in execution of the task set.
  \[ U = \sum_{i=1}^{m} \left( \frac{C_i}{T_i} \right) \]

- Corresponding to a priority assignment, a task set is said to fully utilize the processor if the priority assignment is feasible for the set and if an increase in the run-time of any of the tasks in the set will make the priority assignment infeasible

- For a given fixed priority scheduling algorithm, the least upper bound of the utilization factor is the minimum of the utilization factors over all sets of tasks that fully utilize the processor.
THEOREM 3: For a set of two tasks with fixed priority assignment, the least upper bound to the processor utilization is \( U = 2(2^{1/2} - 1) \).

PROOF:

- Let the two tasks be such that \( T_2 > T_1 \). Therefore task 1 has the higher priority.
- In a critical instant zone of task 2, there are \(|T_2/T_1|\) requests for task 1. Let us now adjust \( C_2 \) to fully utilize the available processor time in the critical zone:
  - **Case 1**: The run-time \( C_1 \) is short enough that all requests for task 1 within the critical zone of \( T_2 \) are completed before the next task 2 request. That is:
    \[ C_1 \leq T_2 - T_1 \lfloor T_2/T_1 \rfloor \]
    Thus the largest value of \( C_2 \) is
    \[ C_2 = T_2 - C_1 \lfloor T_2/T_1 \rfloor \]
    The corresponding utilization factor is
    \[ U = 1 + C_1 \left( \frac{1}{T_1} - \frac{1}{T_2} \right) \lfloor T_2/T_1 \rfloor \]
    In this case \( U \) is monotonically decreasing with \( C_1 \).
  - **Case 2**: The execution of the \( \lfloor T_2/T_1 \rfloor \)'th request for task 1 overlaps with the next request for task 2. In this case
    \[ C_1 \geq T_2 - T_1 \lfloor T_2/T_1 \rfloor \]
    Thus the largest value of \( C_2 \) is
    \[ C_2 = -C_1 \lfloor T_2/T_1 \rfloor + T_1 \lfloor T_2/T_1 \rfloor \]
    and the corresponding utilization factor is
    \[ U = (T_1/T_2) \lfloor T_2/T_1 \rfloor + C_1 \left( \frac{1}{T_1} - \frac{1}{T_2} \right) \lfloor T_2/T_1 \rfloor \]
    In this case \( U \) is monotonically increasing with \( C_1 \).
  - The minimum \( U \) occurs at the boundary between these two cases. That is, for
    \[ C_1 = T_2 - T_1 \lfloor T_2/T_1 \rfloor \]
    Thus the largest value of \( C_2 \) is
    \[ C_2 = T_2 - C_1 \lfloor T_2/T_1 \rfloor \]
    So,
    \[ U = 1 - f(1 - f)/(1 + f) \]
    Minimizing \( U \) over \( f \), we determine that at \( f = 2^{1/2} - 1 \), \( U \) attains its minimum value which is 
    \[ U = 2(2^{1/2} - 1) = 0.83 \]

End of Proof

THEOREM 4: For a set of \( m \) tasks with fixed priority order, and the restriction that the ratio between any two request periods is less than 2, the least upper bound to the processor utilization factor is \( U = m(2^{1/m} - 1) \).

PROOF: Not covered in this course

THEOREM 5: For a set of \( m \) tasks with fixed priority order, the least upper bound on the processor
utilization is \( U = m(2^{1/m} - 1) \).

PROOF: Not covered in this course

---

**Deadline Driven Scheduling Algorithm**

- A task will be assigned the highest priority if the deadline of its current request is the nearest, and will be assigned the lowest priority if the deadline of its current request is the farthest.
- At any instant, the task with the highest priority and yet unfulfilled request will be executed.

**THEOREM 6:** When the deadline driven scheduling algorithm is used to schedule a set of tasks on a processor, there is no processor idle time prior to an overflow (a missed deadline).

**PROOF:**

![Figure: Processing overflow following a processor idle period](image)

- Times of the first request for each of the \( m \) tasks after the processor idle period \([t_1, t_2] \) are denoted \( a \), \( b \), \( c \), ..., \( m \) (in the figure).
- If we were to move, from \( t_2 \) onward, all requests of task 1 up (to the left that is) so that \( a \) will coincide with \( t_2 \). Since there is no processor idle time between \( t_2 \) and \( t_3 \), there will be no processor idle time between \( t_2 \) and \( t_3 \) even after \( a \) is moved up. Moreover, an overflow will occur either at or before \( t_3 \).
- Repeating the above argument one can conclude that, if all tasks are initiated at \( t_2 \), there will be an overflow with no processor idle time prior to it. However, this is a contradiction to the assumption that starting at time 0 there is a processor idle period prior to an overflow.

End of proof

**THEOREM 7:** For a given set of \( m \) tasks, the deadline driven scheduling algorithm is feasible if and only if

\[
(C_1/T_1) + (C_2/T_2) + \ldots + (C_m/T_m) \leq 1
\]
PROOF:

Note: *if and only if* means that the above condition is necessary and sufficient for feasibility.

- To prove necessity:
  - The total demand for computation time by all tasks between \( t = 0 \) and \( t = T \) is:
    \[
    (T_2 T_3 \ldots T_m)C_1 + (T_1 T_2 \ldots T_m)C_2 + \ldots + (T_1 T_2 \ldots T_{m-1})C_m
    \]
  - If the total demand exceeds the available processor time, i.e.,
    \[
    (T_2 T_3 \ldots T_m)C_1 + (T_1 T_2 \ldots T_m)C_2 + \ldots + (T_1 T_2 \ldots T_{m-1})C_m > T_1 T_2 \ldots T_m
    \]
    or
    \[
    (C_1/T_1) + (C_2/T_2) + \ldots + (C_m/T_m) > 1
    \]
    then, there is no feasible scheduling algorithm.

- To prove sufficiency (by contradiction):
  - Assume that the condition
    \[
    (C_1/T_1) + (C_2/T_2) + \ldots + (C_m/T_m) \leq 1
    \]
    is satisfied and yet the scheduling algorithm is not feasible. -- > There is an overflow between \( t = 0 \) and \( t = T_1 T_2 \ldots T_m \).
  - According to theorem 6, there is a \( t = T \) (\( 0 \leq T \leq T_1 T_2 \ldots T_m \)) at which there is an overflow with no processor idle time between \( t = 0 \) and \( t = T \).

Let \( a_1, a_2, \ldots, b_1, b_2, \ldots \), denote request times of the \( m \) tasks immediately prior to \( T \), where \( a_1, a_2, \ldots \) are the request times of tasks with deadlines at \( T \), and \( b_1, b_2, \ldots \) are the request times of tasks with deadlines beyond \( T \). Two cases arise:

- Case 1: None of the computations requested at \( b_1, b_2, \ldots \) was carried out before \( T \).
  - Therefore, the total demand for computation time between \( 0 \) and \( T \) is
    \[
    \lfloor T/T_1 \rfloor C_1 + \lfloor T/T_2 \rfloor C_2 + \ldots + \lfloor T/T_m \rfloor C_m
    \]
  - Since there is no processor idle period,
    \[
    \lfloor T/T_1 \rfloor C_1 + \lfloor T/T_2 \rfloor C_2 + \ldots + \lfloor T/T_m \rfloor C_m > T
    \]
  - Also, since \( x \geq \lfloor x \rfloor \) for all \( x \),
    \[
    (T/T_1)C_1 + (T/T_2)C_2 + \ldots + (T/T_m)C_m > T
    \]
or
\[(C_1/T_1) + (C_2/T_2) + \ldots + (C_m/T_m) > 1\]

Which is a contradiction

**Case 2:** Some of the computations requested at b1, b2,... were carried out before T.

- Since an overflow occurs at T, there must exist a point T' such that none of the requests at b1, b2, ... was carried out within the interval \(T' \leq t \leq T\). In other words, within this interval, only those requests with deadlines at or before T will be executed.
- The fact that one or more of the tasks having requests at the b_i's is executed until \(t = T'\) means that all those requests initiated before T with deadlines at or before T have been fulfilled before T.
- Therefore, the total demand for processor time between \(T' \leq t \leq T\) is less than or equal to
  \[\left\lfloor \frac{T-T'}{T_1} \right\rfloor C_1 + \left\lfloor \frac{T-T'}{T_2} \right\rfloor C_2 + \ldots + \left\lfloor \frac{T-T'}{T_m} \right\rfloor C_m\]

That an overflow occurs at T means that
\[\left\lfloor \frac{T-T'}{T_1} \right\rfloor C_1 + \left\lfloor \frac{T-T'}{T_2} \right\rfloor C_2 + \ldots + \left\lfloor \frac{T-T'}{T_m} \right\rfloor C_m > T - T'\]

or,
\[(C_1/T_1) + (C_2/T_2) + \ldots + (C_m/T_m) > 1\]

Which is a contradiction

**End of Proof**