Real Time Systems: Scheduling Algorithms

Index

Introduction

Various basic scheduling algorithms. For these to work, we need more information about the task other than it's basics. Now each task also has a priority P.

Concepts

New concepts

Hyperperiod

A hyperperiod H is period where the set of tasks will go back to the base line. It is computed as the *least common factor* of all periods. Formally written:

$$H = \operatorname*{lcm}_{1 \leq i \leq n} T_i$$

Secondary Period

A secondary period where it is fitted between the longest computing time, shortest deadline. It has 2 other conditions. Formally defined:

$$\begin{cases} T_s \in [\min_{1 \leq i \leq n} D_i, \max_{1 \leq i \leq n} C_i] \\ H = kT_s, k \in \mathbb{Z}^+ \\ \forall i : 2T_s - \gcd(T_s, T_i) \leq D_i \end{cases}$$

Secondary Period: First Condition

The first condition of the secondary period is:

$$T_s \in [\min_{1 \leq i \leq n} D_i, \max_{1 \leq i \leq n} C_i]$$

Basically it states that T_{s} must be *grater than or equal* to the *maximum compute time* of all tasks. It also needs to be *less than or equal* the *shortest deadline* of all tasks.

Secondary Period: Second Condition

This condition is defined as:

$$H = kT_s, k \in \mathbb{Z}^+$$

This condition states that T_s is proportional to the hyperperiod H with a factor of k, with k being a positive complex ($\mathbb{Z}^+$) value.

Secondary Period: Third Condition

This condition is defined as:

$$\forall i : 2T_s - \gcd(T_s, T_i) \leq D_i$$

It basically means that for all tasks, the operation:

$$2T_s - \gcd(T_s, T_i)$$

Must be *less than or equal* to the deadline of the each task. This condition might fail and you can try *task splitting*. This technique consists on splitting a single task into different parts, always without the *critical section* in the edge of a section.

$$
T_a \text{ is split into 2 parts :}
$$

$$
\begin{cases} T_{a1} & = T_a \\ T_{a2} & = T_a \end{cases}
$$

$$
\begin{cases} D_{a1} & = D_a \\ D_{a2} & = D_a \end{cases}
$$

$$
\begin{cases} C_{a1} & = C_a \\ C_{a2} & = C_a \end{cases}
$$

Response Time Analysis

This analysis checks the interference due to preemption of a higher priority task. Note that this only applies if $D_i \leq T_i$.

The response time R_i is based on the computing time c_i and the interference I_i produced by a higher priority task, preempting this lower priority one. It is defined as:

$$\begin{aligned}
&\forall \tau_i : R_i = C_i + I_i \\
&I_i = \sum_{j \in hp(i)} \lceil \tfrac{R_i}{T_j} \rceil C_j \\
&hp(i) = \{j : 1..n | P_j > P_i\}
\end{aligned}$$

The function $hp(i)$ start for *higher priority* and it is a set of tasks that have higher priority than the current one.

The solution basically works using this function:

$$W_i^{n+1} = c_i + \sum_{j \in hp(i)} \lceil \tfrac{W_i^n}{T_j} \rceil C_j$$

With $W_i^0 = c_i$, corresponding to the response time of task i which cannot be preempted by any other task. Usually this is the task number 1.

It recursively computes the value W until reaching $W_i^{n+1} = W_i^n$, which implies the solution is in a *steady-state* and the response time of task i is $R_i = W_i^n$.

It can also finish when the release time is greater than the deadline for any task.

Critical Time

Critical time refers to instants where a lower priority task is activated at the same time as a higher priority one.

Absolute Deadline

An *absolute deadline* is defined as:

$$d_i = \phi_i + kT_i + D_i$$

The value ϕ_i refers to the initial phase.

Contribution

The contribution of a task refers to how much time is uses of an interval. It is denoted as $\eta_i(t_a, t_b)$.

Processor on Demand Criterion

It checks that at any interval, the required computation by a task set is not greater than the available time. Using response time r and absolute deadline d, it is defined as:

$$g(t_1, t_2) = \sum_{\substack{r_{i,k} \geq t_1 \\ d_{i,k} \leq t_2}} C_i$$

A set of tasks are schedulable if in any time interval the processor demand does not exceed the available time.

$$\forall t_1, t_2 : g(t_1, t_2) \leq t_1 - t_2$$

To test this, it is required to find the instances were the task is contributing between t_1 and t_2. Also taking into account the contribution:

$$g(t_1, t_2) = \sum_{i=1}^{n} \eta_i(t_1, t_2) C_i = \sum_{i=1}^{n} \max(0, \lfloor \frac{t_2 + T_i - D_i}{T_i} \rfloor - \lceil \frac{t_1}{T_i} \rceil) C_i$$

Given that:

$$\begin{cases} \forall i & : \phi_i = 0 \\ t_1 & = 0 \\ t_2 & = L \end{cases}$$

Schedulability is **ensured** if $g(0, L) \leq L$. The formula simplifies to:

$$g(0, L) = \sum_{i=1}^{n} \eta_i(0, L) C_i = \sum_{i=1}^{n} \lfloor \frac{L + T_i - D_i}{T_i} \rfloor C_i$$

If $D_i = T_i$ then it is simplified to:

$$g(0, L) = \sum_{i=1}^{n} \lfloor \frac{L}{T_i} \rfloor C_i$$

Arrival Time

Tasks might *arrive* at a different time than when it's period starts. It is ensured that it's arrival time is fitted:

$$a_i + c_i \leq d_i$$

Aperiodic tasks

An *aperiodic task* is a task that has an arrival time other than 0.

Cyclic Scheduler

A cyclic scheduler is one where the scheduler chooses which task will be executed. This scheduling is composed of 2 elements:

- **Algorithm**: It determines how the tasks are executed
- **Analysis**: It guarantees timing constraints

In these systems, the time constraints are ensured by design. It needs a **time mechanism** that triggers the scheduler periodically.

Cyclic Scheduler: Requisites

The baseline of these systems are defined as:

- 1 processor.
- Static tasks.
- Periodic tasks.
- No *precedence* among tasks.
- The WCET is know for all tasks. They are all less of equal to their deadline.
- Deadlines of each task are *equal* to their period.

Cyclic Scheduler: Methodology

For a *cyclic scheduler* to be schedulable it needs to meet 2 criteria:

- The utilization factor must be *less or equal* than 1
- Using the hyperperiod find the secondary period that satisfies all of it's conditions.

Cyclic Scheduler: Pros

- It is **static**, simple, easy to handle and robust
- Deadlines are ensured by design
- No concurrency nor preemption
- No mutual exclusion
- Low-Level scheduler

Cyclic Scheduler: Cons

- Not flexible
- Segmentation of tasks increases the complexity
- Not adequate for *sporadic tasks*
- Hard to find task allocation within few frames
- Low-Level scheduler

Rate Monotonic

Since each task has different rates (periods) we can order them by that criteria. High frequency tasks have higher priorities. Formally written:

$$\forall \tau_i,\ \tau_j:\ T_i < T_j \ \Rightarrow P_i > P_j$$
$$P_i \propto \frac{1}{T_i}$$

This means that the priorities are proportional to the inverse of the period. The system will run the task with highest priority on each tick. This implies that preemption is possible.

Other important properties:

- The schedulability analysis attempts to know in advance the if all release times occurs before its deadline- The analysis is only performed at the *critical time*.
- At runtime, the scheduler checks at each tick, the task with highest priority and dispatches it.

Rate Monotonic: Requisites

These systems have the same base requisites as a cyclic scheduler. It adds 2 new requirements:

- 1 processor.
- Static tasks.
- Periodic tasks.
- No *precedence* among tasks.
- The WCET is know for all tasks. They are all less of equal to their deadline.
- Deadlines of each task are *equal* to their period.
- **Tasks can be preempted**
- **Real Time kernels uses *fixed priorities***

Rate Monotonic: Methodology

For a rate *monotonic scheduler* to work it needs to fulfill one of 2 *sufficient* conditions:

- Factor utilization factor sum
- Hyperbolic bound

If neither of these 2 conditions are satisfied, check response time analysis for the critical section.

Rate Monotonic: Methodology > Condition 1

This first condition requires the sum of the utilization factor of all tasks needs to be *less than or equal* to $n(\sqrt[n]{2} - 1)$. Formally written:

$$U_{total} = \sum_{i=1}^{n} U_i \leq n(\sqrt[n]{2} - 1)$$

Rate Monotonic: Methodology > Condition 2

This condition checks the following formula:

$$\prod_{i=1}^{n}(U_i + 1) \leq 2$$

Rate Monotonic: Pros

- At design time, priorities are configured based on rate of occurrence of each task
- 2 sufficient conditions
- Response time analysis being a necessary and sufficient condition
- Feasibility of scheduler is checked at critical time (less work to do)

Rate Monotonic: Cons

- It allows *preemption*
- Depends a lot on CPU utilization.

$$\lim_{n\to\infty} U_{\text{total}} = \lim_{n\to\infty} n(\sqrt[n]{2} - 1) = ln(2) = 0.6931...$$

Deadline Monotonic

Deadline monotonic is a variant of rate monotonic, used for tasks with deadlines *less or equal* than their period.

$$\forall \tau_i,\ \tau_j :\ D_i < D_j \ \Rightarrow P_i > P_j$$
$$P_i \propto \frac{1}{D_i}$$

This means that the priorities are proportional to the inverse of the deadline. The system will run the task with highest priority on each tick. This implies that preemption is possible.

Other important properties:

- The schedulability analysis attempts to know in advance the if all release times occurs before its deadline
- The analysis is only performed at the *critical time*.
- At runtime, the scheduler checks at each tick, the task with highest priority and dispatches it.

Deadline Monotonic: Requisites

- 1 processor.
- Static tasks.
- Periodic tasks.
- No *precedence* among tasks.
- The WCET is know for all tasks. They are all less of equal to their deadline.
- Deadlines of each task **are *less or equal*** to their period.
- **Tasks can be preempted**
- **Real Time kernels uses *fixed priorities***

Deadline Monotonic: Methodology

It only applies response time analysis, not using the 2 other sufficient conditions of rete monotonic.

Deadline Monotonic: Pros

- At design time, priorities are configured based on rate of occurrence of each task
- Response time analysis being a necessary and sufficient condition

Deadline Monotonic: Cons

- Preemption
- Performance depends on system ticks

Earliest Deadline First

At run time, each system tick will check the priorities of active tasks and priorities are modified if needed.

$$\forall t, \tau_i, \tau_j : \; d_i < d_j \Rightarrow P_i > P_j$$
$$P_i \propto \frac{1}{d_i}$$

Where d is the absolute deadline. This means that priorities are proportional to the inverse of the absolute deadline. At each system tick, the scheduler looks for the highest priority tasks among active ones. This implies that preemption is possible.

Other important properties:

- The schedulability analysis attempts to know in advance the if all release times occurs before its deadline

Earliest Deadline First: Requisites

- 1 processor
- ~~Static tasks~~
- ~~Periodic tasks~~
- ~~No *precedence* among tasks~~
- The WCET is know for all tasks. They are all less of equal to their deadline
- Deadlines of each task **are *less or equal*** to their period
- **Periodic or aperiodic tasks**
- **It allows precedence among tasks**
- **Tasks can be preempted**
- **Real Time kernels uses *dynamic priorities***

Earliest Deadline First: Methodology

There are 2 cases to check:

1. If $D_i = T_i$ then the necessary and sufficient condition is $U_{total} \leq 1$
2. If $D_i < T_i$ then it is necessary to check processor on demand criterion

Earliest Deadline First: Methodology > Processor on Demand Criterion

To check processor on demand criterion it is required to find L:

$$L \geq \sum_{i=1}^{n} (\lfloor \frac{L - D_i}{T_i} \rfloor + 1) C_i$$

Using this value, find $g(0, L) \leq L$ with the absolute deadlines obtained from the following set:

$$\begin{cases} D & = [d_k | d_k \leq \min(H, L^*)] \\ H & = \text{lcm}(T_1, T_2, ..., T_n) \\ L^* & = \dfrac{\sum_{i=1}^{n}(T_i - D_i)U_i}{1 - U_{total}} \end{cases}$$

Earliest Deadline First: Precedence

One special case of EDF consists of task having precedence, meaning $\tau_a \to \tau_b$. The main effect is that both arrival time and deadlines are modified::

$$\begin{aligned} &\tau_a \to \tau_b : \\ &\qquad \begin{cases} a_b^* & = a_a + C_a \\ d_a^* & = d_b + C_b \end{cases} \\ &\text{Where } a \text{ is the arrival time} \end{aligned}$$

Given that a set J of tasks can transform into a set J^* of tasks, both the arrival times and deadlines must be modified:

$$\begin{aligned} &\text{From the set of tasks } J \text{ to } J^* \\ \\ &\textbf{Arrival time}: \\ &\quad \left| \begin{array}{l} \text{1. Select task } \tau_i \text{ with a modified immediate predecesor} \\ \text{2. } \max_{\text{arrival}} = \max\limits_{\tau_k \to \tau_i}(a_k^* + C_k), k \in J \\ \text{3. Set } a_i^* = \max\{a_i, \max_{\text{arrival}}\} \end{array} \right. \\ &\textbf{Deadline}: \\ &\quad \left| \begin{array}{l} \text{1. Select task } \tau_i \text{ with a modified immediate successor} \\ \text{2. } \min_{\text{deadline}} = \min\limits_{\tau_k \to \tau_i}(d_k^* - C_k), k \in J \\ \text{3. Set } d_i^* = \min\{d_i, \min_{\text{deadline}}\} \end{array} \right. \end{aligned}$$

Earliest Deadline First: New Task

Given that EDF checks at runtime, you can add tasks to it. This will transform the set of tasks J to J_{new}.

For this to work, the following must be true:

$$\begin{aligned} &\text{With } J_{\text{new}} \bigcup J = J' : \\ &\qquad \text{for all } i \text{ in } J' : f_i \leq d_i \\ &\qquad\qquad \begin{cases} f_i = \sum_{k=1}^{n} b_k \\ b_k : \text{ remaining } \textit{Worst Case Execution Time} \text{ of a task } i \text{ in } J' \end{cases} \end{aligned}$$

Basically check if adding this new task, with the remaining times up to the extra time for not reaching the deadline of each tasks, it will fit.

Earliest Deadline First: Pros

- Uses *dynamic priorities* set at runtime
- If $T_i = D_i$, then $U_{total} \leq 1$ is a sufficient condition
- It accepts aperiodic tasks

Earliest Deadline First: Cons

- It is less predictable and less controllable, when trying to reduce response time
- It requires more overhead
- Overload can lead to a domino effect