Priority Based Algorithms and Practical Issues with Real-Time Scheduling

Specifically speaking, a scheduling algorithm is a set of rules that determine the task to be executed at a particular moment. In this document, the scheduling algorithms assign priorities to tasks, and whenever a scheduling decision is to be made, the algorithms select the task with the highest priority as the one to execute.

1 Rate-Monotonic Scheduling algorithm

1.1 The Algorithm

The Rate-Monotonic Scheduling (RMS) algorithm was one of the first scheduling algorithms developed for real-time systems and is still very widely used. The RMS is a static scheduling policy because it assigns fixed priorities to tasks. It turns out that these fixed priorities are sufficient to effectively schedule the tasks in many situations.

The algorithm was proposed under the following assumptions:

A1: All tasks are pre-emptable
A2: Only processing requirements are significant.
A3: All task are non-interacting – all tasks are independent.
A4: All tasks are periodic.
A5: The deadline of a task is equal to its period.

The algorithm can be described as

(1) The higher priority tasks always interrupt the lower priority tasks.

(2) Priorities assigned to tasks are inversely proportional to the length of the tasks' period (higher priority number → higher priority).

1.2 An Example

Scheduling problem:

Consider a system with task details given below.

Task	Execution time (ms)	Period (ms)
T1	20	100
T2	40	150
T3	100	350

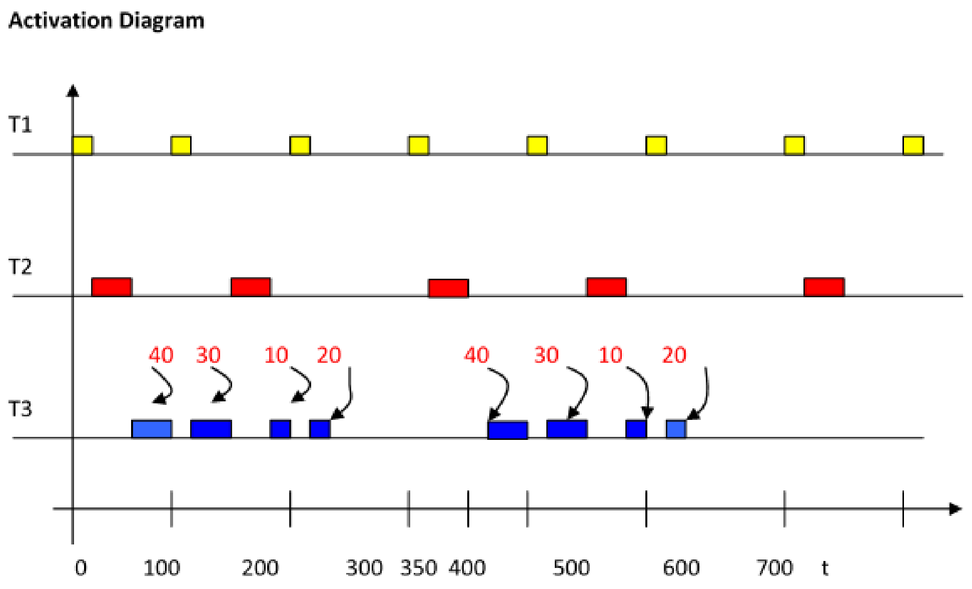
Scheduling result:

Step 1 - Assign the priorities so that they are inversely proportional to the tasks period (shorter period, i.e. the task occurs more often, means higher priority).

Task	Execution time (ms)	Period (ms)	Priority
T1	20	100	1
T2	40	150	2
T3	100	350	3

Priority 1 indicates the highest priority in the table above.

Step 2 - Determine the run-time sequence. The activation diagram below shows the run-time sequence. Since the highest priority task available always run, the activation diagram can be constructed by first scheduling the highest priority task, then the next highest, etc..



1.3 Schedulability analysis

There are several numerical techniques that can be sued to analysis the ability of RMS to schedule a given set of tasks.

1.3.1 Liu and Layland method

There is a simple theoretical result for the analysis of schedulability using RMS. This is the Liu and Layland result which can be described as below:

A set of independent periodic tasks scheduled by RMS always meet their respective deadlines if the following condition is satisfied:

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \le U(n) = n(2^{1/n} - 1)$$

Where,

n : the number of tasks to be scheduled.
C_i : worst case task execution time of task-*i*
T_i : period of task-*i*
U(n): utilisation bound of the n tasks.

Note that the expression $\sum_{i=1}^{n} \frac{C_i}{T_i}$ is called system utilisation, which is one of the key metrics in evaluating a scheduling algorithm. In the previous example, the system utilisation is:

$$\sum_{i=1}^{n} \frac{C_i}{T_i} = 20/100 + 40/150 + 100/350 = 0.753 \equiv 75.3\%$$

This means that in this system the CPU uses 75.3% of its time resource to do useful work.

Applying the Liu and Layland result and comparing the system utilisation with its bound in this case:

$$U(n)\big|_{n=3} = n(2^{1/n} - 1)\big|_{n=3} = 0.779 \equiv 77.9\%$$

This indicates that

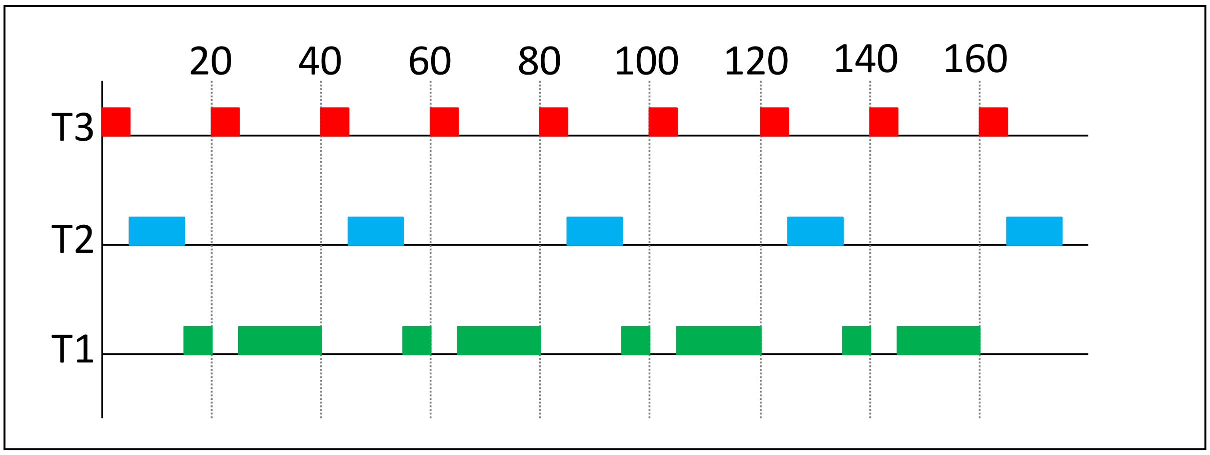
$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq U(n)$$

And therefore the system is schedulable

Note that the Liu and Layland result is a *sufficient, but not necessary, condition*. That is, there may be task sets with a utilisation greater than *U(n)*, but less than or equal to 1 (or 100%), that are still schedulable by the RMS algorithm. For example the next system is schedulable although the condition in the Liu and Layland result is not satisfied.

Task	Execution time	Period	Utilisation(%)
T1	40	80	50
T2	10	40	25
T3	5	20	25
Total utilisation			100

Using an activation diagram to show how the scheduling can be achieved:



1.3.2 Completion time method

The completion time method is more complex than the Liu and Layland method. However, it does give a precise time when each task complete on first execution and thus by comparison to the task periods, exactly determines the ability to schedule the system. The completion time method can be time consuming to calculate by hand, but is easily implemented in software to give results quickly and easily.

Consider a set of *N* tasks T_1 to T_N with computation time C_j and periodicity $T_j, j=1,\dots,N$. Under the following assumptions:

1) Tasks are independent and strictly periodic.
2) Priorities are allowed as : T_1, Priority 1; T_2, Priority 2; …;T_N, Priority N.

the completion time of task n ($1 \le n \le N$) can be estimated using the formula

$$a_{i+1} = C_n + \sum_{j=1}^{n-1} \left\lceil \frac{a_i}{T_j} \right\rceil C_j$$

until $a_{i+1} = a_i$, where

$$a_0 = \sum_{j=1}^{n} C_j$$

and $\lceil\ \rceil$ is a ceiling function. Specifically speaking, if x is not an integer then, $\lceil x \rceil$ is rounded up to the next integer.

To demonstrate how to use the completion time method, consider the system:

	C_i	Ti
T1	5	20
T2	10	40
T3	40	80

<u>The completion time of task T1:</u>

Because

$$a_{i+1} = C_n + \sum_{j=1}^{n-1} \left\lceil \frac{a_i}{T_j} \right\rceil C_j \Bigg|_{n=1} = C_1$$

and

$$a_0 = \sum_{j=1}^{n} C_j \Bigg|_{n=1} = C_1$$

It is known that

$$a_1 = a_0 = C_1 = 5$$

So the completion time of task T1 is 5.

<u>The completion time of task T2:</u>

$$a_{i+1} = C_n + \sum_{j=1}^{n-1} \left\lceil \frac{a_i}{T_j} \right\rceil C_j \Bigg|_{n=2} = C_2 + \left\lceil \frac{a_i}{T_1} \right\rceil C_1$$

and

$$a_0 = \sum_{j=1}^{n} C_j \Bigg|_{n=2} = C_1 + C_2 = 15$$

It is known that $a_1 = C_2 + \left\lceil \frac{15}{20} \right\rceil C_1 = 10 + C_1 = 10 + 5 = 15 = a_0$

So the completion time of task T2 is 15.

<u>The completion time of task T3:</u>

Because

$$a_{i+1} = C_n + \sum_{j=1}^{n-1} \left\lceil \frac{a_i}{T_j} \right\rceil C_j \Bigg|_{n=3} = C_3 + \sum_{j=1}^{3-1} \left\lceil \frac{a_i}{T_j} \right\rceil C_j = C_3 + \left\lceil \frac{a_i}{T_1} \right\rceil C_1 + \left\lceil \frac{a_i}{T_2} \right\rceil C_2$$

and

$$a_0 = \sum_{j=1}^{n} C_j \Bigg|_{n=3} = C_1 + C_2 + C_3 = 55$$

It is known that

$$a_1 = C_3 + \left\lceil \frac{a_0}{T_1} \right\rceil C_1 + \left\lceil \frac{a_0}{T_2} \right\rceil C_2 = 40 + \left\lceil \frac{55}{20} \right\rceil C_1 + \left\lceil \frac{55}{40} \right\rceil C_2 = 40 + 15 + 20 = 75$$

$$a_2 = C_3 + \left\lceil \frac{a_1}{T_1} \right\rceil C_1 + \left\lceil \frac{a_1}{T_2} \right\rceil C_2 = 40 + \left\lceil \frac{75}{20} \right\rceil 5 + \left\lceil \frac{75}{40} \right\rceil 10 = 40 + 20 + 20 = 80$$

$$a_3 = C_3 + \left\lceil \frac{a_2}{T_1} \right\rceil C_1 + \left\lceil \frac{a_2}{T_2} \right\rceil C_2 = 40 + \left\lceil \frac{80}{20} \right\rceil 5 + \left\lceil \frac{80}{40} \right\rceil 10 = 40 + 20 + 20 = 80 = a_2$$

So the completion time of task T3 is 80.

Because 5 < 20 (the deadline for T1), 15 < 40 (the deadline for T2), and 80 = 80 (the deadline for T3), the system is schedulable.

2 Earliest-Deadline-First Scheduling

The Earliest deadline first (EDF) is another well-known scheduling algorithm. It is a dynamic priority algorithm – it changes task priorities during execution based on initiation times. As a result it can achieve higher CPU utilisations than RMS.

The EDF algorithm is also very simple and is based on all the assumptions made for RMS except A4 and A5 since the tasks in the system do not have to be periodic. But, if the tasks happen to be periodic, the deadlines are often also equal to the periods.

2.1 Algorithm

(1) The higher priority tasks always interrupt the lower priority tasks

(2) The task priorities are not fixed but dynamically change. The task is assigned with the highest priority if its deadline is nearest and will be assigned with the lowest priority if the deadline is the furthest.

2.2 An Example

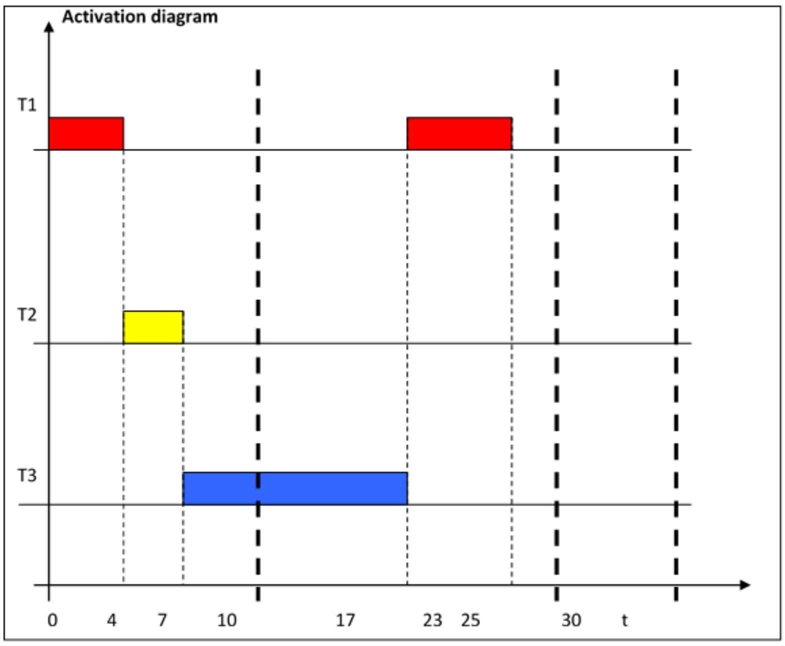
Scheduling problem:

Consider a system with tasks given below.

Task	Arrival time	Execution time	Absolute deadline
T1	0	10	30
T2	4	3	10
T3	7	10	25

Using the EDF algorithm a task can only run when it has data available. As such, the priorities are repeatedly recalculated each time new data arrives or a task completes.

Scheduling result:



2.3 Schedulability analysis

The CPU utilization can be up to 100%. That is, for a given set of *n* periodic tasks with period T_i and execution time C_i, i=1, .., *n*, the EDF algorithm is feasible ***if and only if***:

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$$

2.4 Advantages and disadvantages of dynamic scheduling

Dynamic scheduling is flexible such that it can adapt to an evolving task scenario. The main drawback of a dynamic approach is the significant runtime effort spent in finding a schedule.

The EDF dynamic scheduling algorithm can extract higher utilization out of a CPU. But it may be difficult to diagnose the possibility of an imminent overload. Because the scheduler does take some overhead to make scheduling decisions, a factor that is ignored in the schedulability analysis of both EDF and RMS, running a scheduler at very high utilizations can be somewhat problematic.

In systems with critical deadlines, dynamic scheduling might not be the best choice as it is more difficult to predict what will actually happen in a system. Whereas, static priority scheduling is much more predictable and the run-time sequence can be determined offline, allowing much easier analysis.

3 Some practical issues with real time scheduling

The above analyses of RMS and EDF have made some strong assumptions. These assumptions have made the analyses much more tractable, but the predictions of the analysis may not hold up in practice. Since an incorrect prediction may cause a system to miss a critical deadline, it is important to at least understand the consequences of these assumptions.

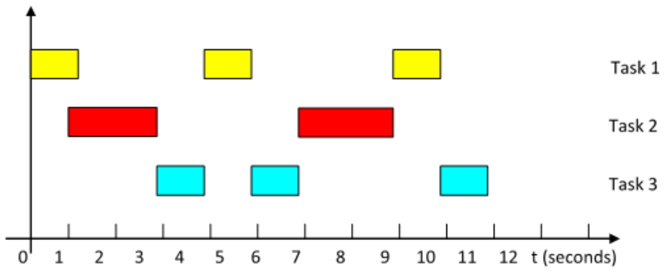
3.1 Data dependency

RMS and EDF all assume that there are no data dependencies between tasks. But this is often not the case in practice and knowledge of data dependencies is important to achieve effective scheduling.

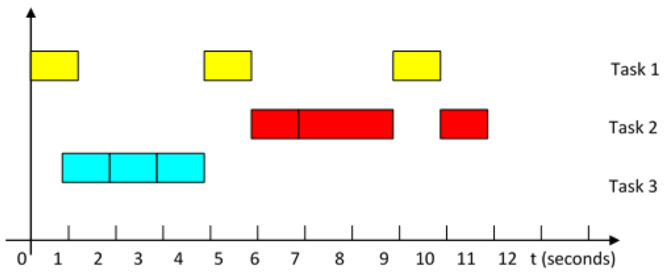
Consider the example below:

Tasks	Execution time	Period
T1	1	4
T2	2	6
T3	3	12

The RMS scheduling result is:



If, in practice, task 2 depends on the output data of task 3, then the scheduling result using the same algorithm is as below.



Consequently task 2 misses its deadline every other period.

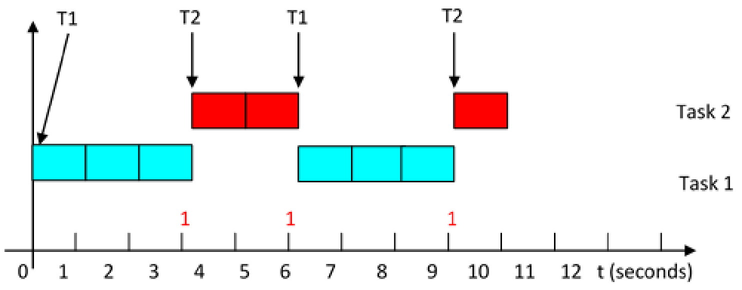
3.2 Context switching

One important simplification we have made above is that contexts can be switched in zero time. On the one hand, this is clearly wrong – we must execute instructions to save and restore context, and we must execute additional instructions to implement the scheduling algorithm. On the other hand, context switching can be implemented efficiently and therefore context switching need not kill performance. The effects of a non-zero context switching time must be carefully analysed for a particular implementation to be sure that the predictions of an ideal scheduling algorithm are sufficiently accurate.

Consider the following example:

Tasks	Execution time	Deadline
T1	3	5
T2	3	10

First, let us try to find a schedule assuming that the context switching time is zero. The following diagram shows a feasible schedule for a sequence of data arrivals that meets all of the deadlines:



Now let us assume that the total time to initiate a task, including context switching and scheduling algorithm evaluation is one time unit. It is easy to see that there is no feasible schedule for the above data arrival sequence, since we require a total of 3+1+2+1+3+1+1=12 time units to execute the one period of T2 and two periods of T1 required every 10 time units to meet the task deadlines (the time in red is for the context switch).

In this example, the overhead is a large fraction of the task execution times and of the periods. In most real-time operating systems, a context switch only takes a very small amount of time (a few microseconds or less), with only slightly more overhead for a simple real time scheduler like RMS. These small overhead times are less likely to cause serious scheduling problems.